THÈSE DE DOCTORAT

École doctorale
« Sciences et Technologies de l'Information et de la Communication »
de Nice - Sophia Antipolis
Discipline Informatique

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
FACULTÉ DES SCIENCES

UNE ARCHITECTURE DE SÉCURITÉ HIÉRARCHIQUE, ADAPTABLE ET DYNAMIQUE POUR LA GRILLE

---

par

Arnaud CONTES

Thèse dirigée par **Denis CAROMEL** *et* **Isabelle ATTALI**

réalisée au sein de l'équipe OASIS
équipe commune de l'I.N.R.I.A. Sophia Antipolis et du laboratoire I3S

présentée et soutenue publiquement le 9 Septembre 2005 à l'E.P.U. devant le jury composé de

Président du Jury	Frank LEPREVOST	Université du Luxembourg
Rapporteurs	Jean-Louis ROCH	INRIA Rhône-Alpes
	Hervé GUYENNET	Université de Besançon
Examinateurs	Bruno MARTIN	I3S - Université de Nice
	Yves ROUDIER	Institut Eurecom
Invité Industriel	Anne HARDY	SAP Labs
Directeur de thèse	Denis CAROMEL	Université de Nice - Sophia Antipolis, Institut Universitaire de France

A Virginie,
A mes parents.
–Arnaud

Table des matières

Table des figures

Liste des tableaux

Remerciements

Mes remerciements vont en premier vers Denis Caromel. Il m'a donné l'opportunité de réaliser cette thèse, tout d'abord en recherchant un financement puis en acceptant d'être mon directeur de thèse. J'ai particulièrement apprécié nos interminables discussions sur ce que devrait être la sécurité des systèmes distribués en général et au sein de la bibliothèque *ProActive*, en particulier. Les principaux résultats présentés dans cette thèse sont issus de ces discussions.
Merci à Françoise Baude pour son soutien et son attention tout au long de cette thèse. Ses conseils et ses commentaires auront été fort utiles.

Je tiens également à remercier Anne Hardy, Frank Leprevost, Hervé Guyennet, Jean-Louis Roch, Yves Roudier et Bruno Martin pour avoir accepté de faire partie du jury et pour leurs remarques pertinentes sur le manuscrit.

J'adresse un grand merci à tous les membres de l'équipe OASIS anciens et présents, stagiaires, doctorants, ingénieurs, permanents, assistant(e)s de projet que j'ai pu croiser au cours de mon séjour dans l'équipe. Toutes ces personnes ont participé à créer et maintenir au sein de l'équipe, une ambiance chaleureuse et studieuse qui va énormément me manquer.
Parmi toutes ces personnes, il y en a à qui je voudrais adresser spécialement quelques mots.

Laurent B. a découvert *ProActive* en même temps que moi lors de notre stage de DEA et a lui aussi été séduit par le côté obscur de la *Force*. Nous avons partagé de nombreux moments forts en émotions diverses allant de la désillusion à la joie du succès de nos travaux de recherche. Notre amitié s'est renforcée au cours de ces épreuves et durera, je l'espère, longtemps après la fin de nos thèses.

Fabrice H. et Julien V. ont joué le rôle de grands frères de thèse en me guidant lors de mes débuts. Ils ont habilement été secondés dans ce rôle par Carine C. et Ludovic H. Merci à vous pour tous vos conseils.

Il y a ensuite tous les doctorants plus ou moins jeunes qui ont partagé cette aventure : Christian B. "l'allemand", Olivier N., Laurent Q., Christian D. , le "clan des chiliens" comprenant Tomas B. et Javier B., Felipe L. "le Mexicain", Alexandre DC "l'homme à la pomme", Guillaume C. qui est destiné à me remplacer dans le rôle de "l'homme qui murmurait à l'oreille des pingouins". Bonne chance à tous pour vos futures thèses, et pour ceux qui l'ont déjà, bonne chance dans vos futurs métiers.

Aux ingénieurs de l'équipe, Lionel M., Romain Q., Igor R., je leur adresse mes remerciements pour leur disponibilité et leur aide.

Les personnes ayant relues ma thèse, tâche fastidieuse : Françoise B., Denis C., Fabrice H. et surtout Virginie L. qui l'a relue depuis les premières versions jusqu'à la version finale, à force de la relire, Virginie doit maintenant maîtriser le sujet aussi bien que moi.

N'oublions *ProActive*, son code source et les joies de la programmation distribuée.

Claire S. devrait recevoir le titre de docteur en assistante de projet. Elle est d'une efficacité

redoutable pour venir à bout des divers formulaires administratifs et simplifier pour autant la vie de doctorant.

Je tiens aussi à remercier les deux personnes sans qui je ne serai sûrement pas en train de finir d'écrire ces remerciements. Ils ont eu la patience de m'élever, de me supporter, de me conseiller et de me motiver jour après jour, année après année. Je veux bien évidement parler de mes parents, Richard et Danièle. je n'oublie pas mon frère Cédric dont la présence pendant toutes ces années a également contribué à faire de moi ce que je suis aujourd'hui.
Le tableau de famille ne serait pas complet sans remercier Virginie qui m'a supporté avec amour tout au long de ce périple.

Je ne peux terminer sans avoir une pensée pour Isabelle et ses deux enfants Ugo et Tom. Leur disparition tragique lors du tsunami qui a frappé le Sri Lanka en décembre 2004 nous a tous terriblement affectés. Isabelle co-encadrait ma thèse avec Denis ; je lui suis extrêmement reconnaissant pour sa constante disponibilité ainsi que pour la rigueur de raisonnement scientifique qui m'a permis de structurer mes idées. La qualité que j'admirais le plus chez Isabelle était la quantité d'énergie qu'elle était capable de déployer en toute circonstance et surtout la capacité de transmettre cette énergie aux personnes qui l'entouraient.
Isabelle, sans ton aide, cette thèse ne serait pas ce qu'elle est, merci pour tout.

—
Arnaud

Chapitre 1

Introduction

Lors d'un discours en 1965, Gordon Moore, un des présidents d'Intel, fit une remarque qui reste toujours d'actualité : selon lui, le nombre de transistors des processeurs devrait doubler tous les 18 mois et permettre ainsi une croissance exponentielle régulière des performances. Cette loi s'est vérifiée au fil du temps. Nous sommes arrivés à un point où la puissance de calcul d'un ordinateur actuel est généralement surdimensionnée par rapport à la charge de travail que lui imposent ses utilisateurs. La mise à disposition à des tiers des ressources inexploitées d'un parc d'ordinateurs est la suite logique de ce constat. La baisse constante du prix du matériel informatique favorise aussi la multiplication du parc matériel au sein des entreprises, des laboratoires de recherches ou chez les particuliers.

1.1 Problématique et objectifs de la thèse

Les *grilles de calcul* permettent de regrouper toute cette puissance afin de l'utiliser. Sous le terme grille de calcul, nous rassemblons tous les outils permettant la globalisation des ressources ainsi que leur mise à disposition.

La mise à disposition des ressources entraîne de manière directe le besoin de sécuriser cet ensemble afin de le protéger contre une utilisation frauduleuse. Ce contrôle s'applique à la fois sur les ressources fournies mais aussi sur les utilisateurs de celle-ci. Le rôle premier d'un mécanisme de sécurité consiste à protéger l'accès aux ressources et de n'accorder l'accès qu'aux utilisateurs autorisés.

Les solutions de grilles de calcul clé en main se sont avérées cruciales dans l'adoption massives des technologies liées aux grilles de calcul par tous les acteurs. Ces solutions permettent de mettre facilement en place des grilles de calcul et répondent majoritairement aux besoins des utilisateurs. Une grille de calcul peut être composée de grappes (*cluster*), de machines parallèles ou encore d'ordinateurs de bureau. Un grappe est composée d'un ensemble de machines dédiées à la grille interconnectées par des réseaux haut-débit. Cependant, si le déploiement d'une application utilisant une seule grille ou un ensemble de grilles similaires (mêmes technologies) reste dans la limite du possible, l'utilisation d'un ensemble de grilles hétérogènes est nettement moins évidente. La première critique qui peut être formulée concerne le manque d'interopérabilité des diverses technologies y compris celles sur lesquelles se reposent les mécanismes de sécurité. En effet, si on s'intéresse au déploiement d'une application, on constate qu'il existe de nombreuses manières de soumettre des tâches ou d'accéder aux ressources. Cette diversité impose une configuration spécifique de l'application pour chaque solution. Suite à la mise en avant de ces problèmes, il est important de proposer une couche logicielle qui permette de découpler la partie contenant la logique de l'application de la partie concernant le processus de déploiement de cette application.

Dans le même ordre d'idées, il apparaît indispensable de pouvoir exprimer la configuration de sécurité d'une application en dehors du code de cette dernière. Cette approche permet ainsi de pouvoir configurer la sécurité d'une application en fonction de son déploiement. Cette solution nous semble tout particulièrement adaptée lors du déploiement d'applications sur plusieurs

grilles de calcul. En effet, la sécurité au sein de la grille de calcul est généralement assurée par les administrateurs de cette dernière. Il peut aussi arriver que deux grilles de calcul puissent être configurées pour fonctionner ensemble de manière sécurisée. Cette configuration suppose la mise en place d'une *organisation virtuelle*. Du point de vue de la sécurité, elle permet de réunir un ensemble d'utilisateurs et de ressources au sein d'une même communauté et d'assurer un certain niveau de confiance à tous ses membres. Si l'organisation virtuelle facilite la création de communautés, sa configuration ne dépend pas d'un utilisateur et peut ne pas répondre à ses besoins immédiats. Dans la plupart des configurations que nous avons rencontrées lors de l'utilisation de plusieurs grilles de calcul, elles sont interconnectées par des réseaux publics comme Internet. La sécurité des communications entre ces grilles n'est pas assurée.

Il nous semble important de garder un modèle de sécurité qui permette, comme dans la plupart des solutions clés en main, aux administrateurs de configurer les paramètres de sécurité de leur grille mais aussi de laisser l'utilisateur exprimer ses propres besoins en termes de sécurité concernant le déploiement de son application.

Cependant, l'objectif de cette thèse est d'incorporer dans ce modèle de sécurité un moyen permettant à tous les acteurs d'exprimer leurs besoins en terme de sécurité. Parallèlement, ce modèle doit être capable de fournir une infrastructure de sécurité au sein de laquelle des applications sans notion explicite de sécurité pourront être déployées de manière sécurisée grâce à l'utilisation de mécanismes de sécurité transparents.

Les travaux présentés dans cette thèse ont été effectué dans le cadre de l'intergiciel (*middleware*) *ProActive*. Il représente une couche logicielle qui s'intercale entre le code applicatif (software) et la partie matérielle (hardware). Son but est de fournir des services facilitant le développement, la composition et l'exécution d'applications dans un environnement réparti.

La bibliothèque cible la programmation et le calcul distribués, son but est de fournir les mécanismes, concepts, services permettant la distribution des divers composants d'une application et cela même si l'application n'avait pas été écrite à l'origine pour fonctionner de manière distribuée. Une application déployée avec la bibliothèque va pouvoir utiliser tout aussi bien des grilles de calcul que des ordinateurs de bureau pour mener à bien son exécution. L'acquisition de toutes ces ressources peut se faire par le biais de plusieurs méthodes de réservations (OAR, LSF, GRAM, etc) ou par connexion directe (RSH, SSH) avec la machine hôte. Les connexions entre les différentes ressources peuvent utiliser plusieurs types de protocoles réseaux tels que RMI, HTTP ou encore la combinaison des protocoles RMI et SSH qui permet l'encapsulation des flux RMI au travers de connexions SSH.

La gestion des ressources est transparente au code applicatif qui s'exécute au dessus de la bibliothèque. Le code utilise ces ressources comme si ces dernières étaient locales et disponibles sur une seule machine. Cette transparence vis-à-vis de la distribution des composants de l'application pose bien évidemment des problèmes de sécurité variables, et fonction du déploiement, voire de la dynamique de l'application.

De plus, il faut noter que la bibliothèque *ProActive* est prévue pour être déployée au-dessus de n'importe quelle machine virtuelle java qui propose les fonctionnalités nécessaires à son exécution. La bibliothèque ne requiert pas l'utilisation d'une machine virtuelle modifiée. Cette approche permet à la bibliothèque d'utiliser n'importe quelle machine virtuelle et contribue ainsi à son utilisation sur tout type d'ordinateur comportant une machine virtuelle java. Ce critère nous oblige à considérer les hôtes sur lequel la machine virtuelle va s'exécuter comme de confiance.

La sécurité est une fonctionnalité clé dans le monde des systèmes distribués. Ces systèmes, du fait de leur distribution, sont beaucoup plus fragiles et sensibles aux problèmes de sécurité. Il est important de concevoir une architecture capable à la fois de supporter les contraintes des systèmes distribués et qui sera en mesure de répondre à l'attente des divers acteurs qu'ils soient développeurs, utilisateurs ou administrateurs. Afin de répondre à tous ces besoins et attentes, notre mécanisme de sécurité doit atteindre les objectifs suivants :

- *Passage à l'échelle* : un des défis des applications distribuées consiste à réunir le nombre maximal de ressources disponibles afin de mener à bien leur exécution en un minimum de temps. Il est possible d'énumérer toutes les interactions possibles dans un système distribué comportant peu d'éléments et d'associer à des règles de sécurité à chacune de ces interactions. Il semble plus improbable de réussir à énumérer toutes ces interactions de sécurité lorsque le système comporte plusieurs centaines voire milliers d'entités qui interagissent entre elles et notamment si l'application ou le système distribué sont capables d'évoluer dynamiquement. Le mécanisme de sécurité doit :
  - permettre d'exprimer des règles de sécurité sur une interaction précise ;
  - introduire des mécanismes permettant de limiter le nombre de règles à écrire sans nuire pour autant à la sécurité de l'application.

- *Hiérarchie* : Une approche hiérarchique permet un partitionnement des ressources dans des ensembles distincts qui seront soumis aux mêmes politiques de sécurité. Cette classification des éléments du système permettra aussi d'écrire des règles de sécurité plus génériques dont les sujets seront les ensembles bien qu'au final les règles s'appliqueront sur les éléments de ces ensembles. Ainsi si une même politique de sécurité doit s'appliquer à toutes les machines d'une grappe de calcul, il doit être possible de n'écrire qu'une seule règle plutôt qu'une règle pour chacune des machines de la grappe de calcul.

- *Transparence* : Même si la sécurité est une des fonctionnalités les plus demandées dans les applications distribuées, elle reste cependant souvent mise à l'écart dans le processus de développement des applications distribuées. Il existe plusieurs raisons notamment dues :
  - au manque de connaissances que peuvent avoir les développeurs ou les utilisateurs des principes liés à la sécurité en général et à la sécurité distribuée en particulier ;
  - à l'inadéquation entre les mécanismes de sécurité proposés et les besoins des applications réparties ;
  - à la difficulté de mise en place des infrastructures de sécurité nécessaires.

  C'est pour ces raisons que nous voulons traiter les concepts et principes de sécurité de manière orthogonale au code applicatif. En utilisant la sécurité de manière transparente, nous allons pouvoir laisser les développeurs continuer à se concentrer sur le développement de leurs applications.

- *Adaptation* : Lorsqu'un objet va être hébergé par une machine donnée, il va être soumis à la politique de sécurité imposée par son hôte. Mais cet objet peut aussi posséder ses propres règles de sécurité. Lors d'une interaction, plusieurs règles de sécurité peuvent venir s'appliquer. Ces règles peuvent être imposées par l'objet lui-même, par l'hôte ou par d'autres entités. La règle de sécurité finale qui s'appliquera sur une interaction donnée sera le résultat d'un calcul portant sur un ensemble de règles de sécurité.

- *Décentralisation* : Les services qu'offrent les systèmes distribués tendent à être implantés de façon totalement décentralisée. Aucun objet du système ne connaît l'état global de celui-ci. Chaque objet doit prendre des décisions en fonction des informations disponibles localement. Il doit en être de même pour notre modèle de sécurité qui doit pouvoir prendre une décision de sécurité concernant un objet local sans pour autant avoir une vision globale de l'architecture de l'application.

- *Evolutivité* : Le système doit être capable de fournir les fonctionnalités de base en termes de sécurité mais aussi fournir un ensemble d'interfaces et de modules bien définis afin d'intégrer et de gérer non seulement les fonctionnalités existantes mais aussi celles qui seront introduites par la suite.

- *Multi-utilisateurs* : Une application distribuée peut être utilisée par plusieurs utilisateurs à la fois. Tous ces utilisateurs peuvent avoir des droits différents, certains étant restreints dans leurs possibilités, les autres non.

– *Dynamicité* : L'environnement dans un système distribué est dynamique et évolue au cours du temps. De nouvelles ressources peuvent devenir disponibles et l'application doit être en mesure des les utiliser. Il faut que cette utilisation se fasse sans affaiblir ou invalider la politique de sécurité initiale. De même, une application distribuée initialement lancée sans sa sécurité activée doit pouvoir aussi interagir avec une autre application dont le système de sécurité est activé et pour laquelle les communications doivent être chiffrées.

– *Tolérance aux pannes* : Du fait de sa répartition, une application peut perdre temporairement ou définitivement le contact avec une partie d'elle-même sans pour autant que son fonctionnement global ne soit remis en question. Il doit en être de même pour notre mécanisme qui doit pouvoir être tolérant aux pannes qui n'entraînent pas l'arrêt de l'application. Notre but n'est pas de créer un mécanisme de sécurité entièrement tolérant aux pannes mais plutôt un mécanisme qui soit capable de gérer un certain nombre de pannes courantes (perte de la connexion, perte d'une partie de l'application) sans que cela n'affecte la sécurité de l'application.

– *Efficacité* : Authentifier et/ou chiffrer des échanges de messages entraîne un impact sur les performances d'une application. Notre mécanisme de sécurité doit pourvoir être adapté à chaque utilisation afin de minimiser le surcoût induit.

1.2 Structure du manuscrit

Ce manuscrit est découpé en plusieurs grandes parties qui reflètent la démarche suivie tout au long de cette thèse.

Tout d'abord, la première partie (*Chapitre 2*) nous permet d'exposer les besoins en terme de sécurité que rencontrent les applications distribuées. Nous continuons par la mise en avant des diverses solutions de sécurité existantes dont nous nous sommes inspirés pour élaborer notre modèle de sécurité afin qu'il réponde aux objectifs énumérés précédemment.

Le chapitre 3 présente un ensemble représentatif d'intergiciels possédant des primitives de sécurité. Nous exposons, tout d'abord, une vue d'ensemble de l'intergiciel avant d'exposer les principes et solutions de sécurité qui le composent. Enfin nous évaluons son niveau d'adéquation par rapport à nos préoccupations.

Nous nous sommes ensuite attachés à montrer le fonctionnement global de la bibliothèque *ProActive* ainsi que sa philosophie (chapitre 4) pour aboutir au chapitre 5 qui présente le modèle de sécurité développé lors de cette thèse. Il est découpé en plusieurs parties complémentaires mais distinctes. Nous commençons par introduire l'idée initiale qui a permis la mise au point d'un modèle de sécurité générique et hiérarchique. Nous présentons ensuite le langage de politiques de sécurité qui a été développé afin de pouvoir exprimer toutes la flexibilité introduite par notre modèle. La dernière partie de ce chapitre expose le concept de propagation dynamique du contexte de sécurité d'une application mettant en évidence l'intérêt de notre approche dans la gestion transparente de la sécurité.

L'implantation du mécanisme de sécurité au sein de la bibliothèque *ProActive* est présentée dans le chapitre 6. Ce chapitre est consacré à montrer les interactions entre notre module de sécurité et les autres fonctionnalités de la bibliothèque. Cette partie est destinée à exposer l'adaptabilité du modèle et son intégration aisée avec les autres mécanismes de la bibliothèque.

Le chapitre 7 présente des tests de performances montrant l'impact du mécanisme de sécurité sur les communications d'une application et leurs répercussions sur la durée d'exécution de l'application.

Finalement le chapitre 8 conclut et présente les perspectives ouvertes par les travaux présentés dans cette thèse.

Chapitre 2

Théories et Modèles

Ce chapitre nous permet, dans un premier temps, d'introduire les notions nécessaires sur la sécurité informatique, il présente ensuite les concepts que nous allons être amenés à utiliser au sein de notre modèle.

2.1 Concepts de base sur la sécurité informatique

Avant toute chose, il convient de définir ce qu'est la sécurité informatique, quels sont ses tenants et ses aboutissants. La définition généralement admise est la suivante :

Définition 1 *La sécurité informatique*
La sécurité d'un système informatique a pour mission principale la protection des informations et des ressources contre toute divulgation, altération ou destruction. L'accès à ces ressources doit être également protégé et un accès autorisé à ces ressources ne doit pas être refusé. La sécurité informatique consiste à utiliser tous les mécanismes disponibles pour garantir les propriétés suivantes : la confidentialité, l'intégrité et la disponibilité.

Toute sécurité est généralement formulée suivant un système composé d'un certain nombre d'entités.

- Les *sujets* ou *principaux* : entités actives manipulant de l'information ;
- Les *objets* : entités contenant de l'information, potentiellement protégées et sur lesquelles un sujet peut ou ne peut pas agir. A chaque objet est associée une liste d'opérations définissant les sujets qui sont autorisés à y accéder.

Il faut noter qu'une entité donnée peut assumer à la fois le rôle d'un sujet si on s'intéresse aux informations que cette entité peut manipuler ou bien le rôle d'un objet au regard des informations qu'elle contient.

Les notions de sujets et d'objets permettent de définir en terme d'accès aux objets du système informatique les propriétés suivantes :

- l'*authentification* est la propriété qui permet de garantir qu'un certain sujet est correctement identifié ;
- la *non-répudiation* est la propriété apportant une preuve indéniable qu'un objet a bien été émis par un sujet ;
- le *contrôle d'accès* permet d'assurer que l'accès à des données par des sujets peut être contrôlé ;
- la *confidentialité* des données est la propriété garantissant qu'une information contenue dans un objet ne sera pas révélée, ni rendue accessible à des sujets non autorisés. Cela signifie que le système informatique doit empêcher les sujets de lire une information confidentielle s'ils n'y sont pas autorisés, mais aussi, qu'il doit empêcher les sujets autorisés à lire une information de la divulguer à d'autres utilisateurs (sauf autorisation). Ce deuxième point est le plus souvent négligé car il est beaucoup plus difficile à assurer. Il s'agit de faire du contrôle de flux afin de s'assurer de la non divulgation d'une information.

- L'*intégrité* est la propriété d'une information à ne pas être altérée. Cela signifie que le système informatique doit empêcher :
  - toute modification de l'information par des sujets non autorisés ;
  - une modification incorrecte par des sujets autorisés ;
  - une modification malicieuse par un sujet authentifié et qui abuse de ses droits (ou qui les a usurpés) ;
  - prévenir qu'un sujet puisse empêcher la modification d'une information.
- La *disponibilité* est la propriété d'une information ou d'un service à être disponible quand un sujet en a besoin. Pour cela, le système informatique doit fournir l'accès à l'information ou au service afin que les sujets autorisés puissent y accéder. Il doit aussi garantir qu'aucun sujet ne puisse empêcher un sujet autorisé à accéder à l'information ou au service.

2.2 Vulnérabilités des systèmes informatiques

Une *attaque* est l'exploitation d'une faille d'un système informatique (système d'exploitation, logiciel) ou bien même de l'utilisateur à des fins non autorisées par le propriétaire du système et généralement répréhensibles.

Les motivations des attaques peuvent être de différentes sortes :
- obtenir un accès au système pour en faire une machine capable d'attaquer d'autres machines (attaque par rebond) ;
- voler des informations : secrets industriels, informations personnelles sur un utilisateur, récupérer des données bancaires, s'informer une entreprise, ... ;
- empêcher le bon fonctionnement d'un service.

Avant de parler des problèmes de sécurité au sein des applications distribuées, nous allons commencer par nous intéresser aux problèmes de sécurité qui existent sur un seul ordinateur.

Un ordinateur est composé d'un ensemble d'éléments matériels et logiciels permettant à une application de pouvoir s'exécuter (figure 2.1). En terme de sécurité, chacun de ces éléments peut être considéré comme le maillon d'une chaîne qui représenterait la sécurité de l'ensemble. La solidité d'une chaîne dépend de la solidité du maillon le plus faible de cette chaîne. Il en est de même pour les ordinateurs.

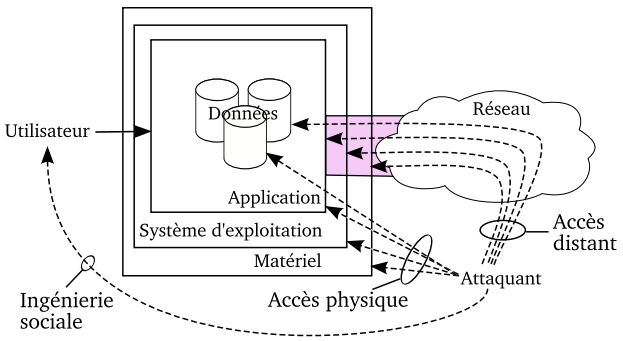


FIG. 2.1 – Divers types d'attaques possibles

Un ordinateur peut être attaqué par plusieurs endroits, nous pouvons les classer selon les critères suivants :
- *Accès physique à la machine.*
  Si un attaquant possède un accès direct sur une machine, il lui est alors virtuellement possible de faire tout ce qu'il veut : récupérer le disque dur, installer de nouvelles cartes d'extension, analyser la mémoire de l'ordinateur à la recherche de clé de chiffrement, modifier le système d'exploitation, les logiciels, ...

– *Ingénierie sociale*.
  L'attaquant obtient un bien ou une information en exploitant la confiance mais également l'ignorance ou la crédulité des utilisateurs ou de tierces personnes. Il s'agira d'exploiter le facteur humain, qui peut être considéré comme le maillon faible de tout système de sécurité. Kevin Mitnick, célèbre notamment pour avoir accédé illégalement aux bases de données des clients de Bell et aux systèmes du Pentagone, de la NASA et de l'US Air Force a théorisé et popularisé cette pratique dans son livre "*The Art Of Intrusion : The Real Stories Behind The Exploits Of Hackers, Intruders and Deceivers*" [88].
– *Accès par le réseau*.
  Lorsqu'une machine est connectée à un réseau, elle doit être considérée comme cible potentielle d'une attaque provenant de n'importe quelles autres machines également connectées à ce réseau, ceci est encore plus vrai lorsque la machine possède un accès à internet. Il faut alors considérer toutes les machines connectés à internet comme des attaquants potentiels. Les moyens et les types d'attaques par le réseau sont nombreux et variés : dénis de service, intrusions, virus, vers et chevaux de Troie, balayage de ports (*ports scan*), altération des messages de et vers des machines, détournement de session TCP, usurpation d'identité (*man in the middle*), attaque par rejeu,. . . .

Le processus de sécurisation d'un système informatique consiste essentiellement à trouver et à supprimer toutes les failles d'un système afin de rendre les attaques inopérantes. Notons que si un administrateur doit combler toutes les failles de ces systèmes, l'attaquant lui n'a qu'à trouver une seule faille pour compromettre la sécurité totale d'un système.

Vulnérabilités des applications distribuées

De manière générale, une application distribuée est composée de plusieurs parties s'exécutant sur plusieurs ordinateurs et échangeant des messages contenant des données pour mener à bien une tâche.

Lorsque l'attaquant ne dispose pas d'accès physique à la machine, il est alors obligé d'obtenir des informations via le réseau interconnectant les diverses parties de l'application. Les attaques dont il dispose alors pour parvenir à ses fins (figure 2.2) sont les suivantes :

– *Interruption* : Une partie de l'application distribuée est détruite ou est devenue inaccessible. Il s'agit d'une attaque sur la disponibilité.
– *Interception* : Un tiers non autorisé intercepte des données. Il s'agit d'une attaque sur la confidentialité.
– *Fabrication* : Un tiers non autorisé insère des données contrefaites dans les communications de l'application. Il s'agit d'une attaque sur l'authentification.
– *Modification* : Un tiers non autorisé intercepte des données et les modifie avant des les envoyer au destinataire. Il s'agit d'une attaque sur l'intégrité.

Les systèmes à agents mobiles

Le paradigme du code mobile [48, 57, 53] est une notion présente depuis le début des années 1990 au sein des systèmes distribués. Un agent mobile est une entité logicielle mobile composée de code, de données et d'un état. Lorsqu'un agent mobile se déplace de machine en machine, il est crucial de s'assurer que l'agent sera exécuté fidèlement et complètement sur chacun des nœuds qu'il visite. Par exemple, dans le cas d'un agent mobile qui visite un certain nombre de sites de commerce électronique afin d'en trouver le meilleur pour un produit donné, nous voulons empêcher les hôtes de l'agent de le modifier de manière à faire apparaître leur offre comme la meilleure. Dans le même ordre d'idée, il faut pouvoir garantir que lors de son transfert d'un hôte vers un autre, l'agent ne sera pas modifié par une entité tierce. Les champs d'application typiques pour ce type de modèle sont, par excellence, l'administration des réseaux [27], la collecte d'informations sur divers sites ou encore le calcul distribué.

Il existe de nombreux systèmes à agents mobiles : Ajanta [63, 64], Telescript [115], les Aglets [118, 62], Mobile Agent Plateform (MAP) [100, 99]. La mobilité du code et des données inhérentes

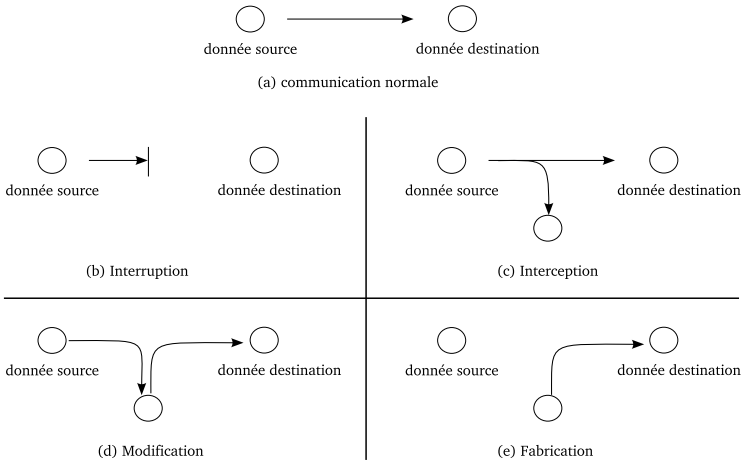


FIG. 2.2 – Attaques sur les communications d'une application distribuée

au modèle des agents mobiles pose bien évidemment des problèmes de sécurité. Harrison [23] présente la sécurité au sein des systèmes à agents mobiles comme l'obstacle le plus important vis-à-vis de leur adoption. William M. Farmer, Joshua D. Guttman et Vipin Swarup [39] exposent les problèmes et les besoins en terme de sécurité des systèmes à agents mobiles génériques, et ils proposent un modèle d'authentification [38].

Hôte de confiance

La sécurité des systèmes distribués, notamment lorsqu'elle inclue des protocoles cryptographiques, est une tâche difficile car elle implique :

- le déploiement des mécanismes de sécurité sur les ordinateurs accueillant les applications distribuées ;
- des processus de création et distribution des secrets partagés par les tiers en présence ;
- un comportement imprévisible des protocoles de communication.

La figure 2.3 représente, dans ses grandes lignes, la classification OSI et permet de situer distinctement la partie dans laquelle se situe les intergiciels (dont *ProActive*) des parties inférieures sur lesquelles un intergiciel n'a aucun contrôle. Dans ce modèle en couche, le niveau de confiance d'un niveau ou sa fiabilité dépendent directement du niveau de confiance ou la fiabilité prouvée ou estimée des couches inférieures. Ainsi le niveau de sécurité d'une application écrite avec un intergiciel donné dépend non seulement de la fiabilité des mécanismes de sécurité fournis par cet intergiciel mais également de la fiabilité en terme de sécurité des couches de l'architecture sous-jacente qui permettent l'exécution de l'application.

Ainsi, l'utilisation de protocoles cryptographiques au niveau applicatif est inutile s'il est impossible de garantir la non-divulgation du secret, qui comme son nom l'indique doit rester secret, par les niveaux inférieurs.

L'hypothèse d'un hôte de confiance, qui va exécuter leur code correctement, est faite par tous les intergiciels orientés applications distribuées offrant des mécanismes de sécurité.

2.3 Le contrôle d'accès

Le terme "accès" suggère qu'un sujet essaie d'accéder à un objet. Son principe repose sur l'utilisation d'un moniteur de référence (figure 2.4) qui est l'élément du système par lequel passent toutes les requêtes d'accès aux objets protégés, et qui décide d'accorder ou non l'accès en fonction de règles de sécurité. Bien entendu, il est fondamental qu'il ne soit pas possible de contourner le

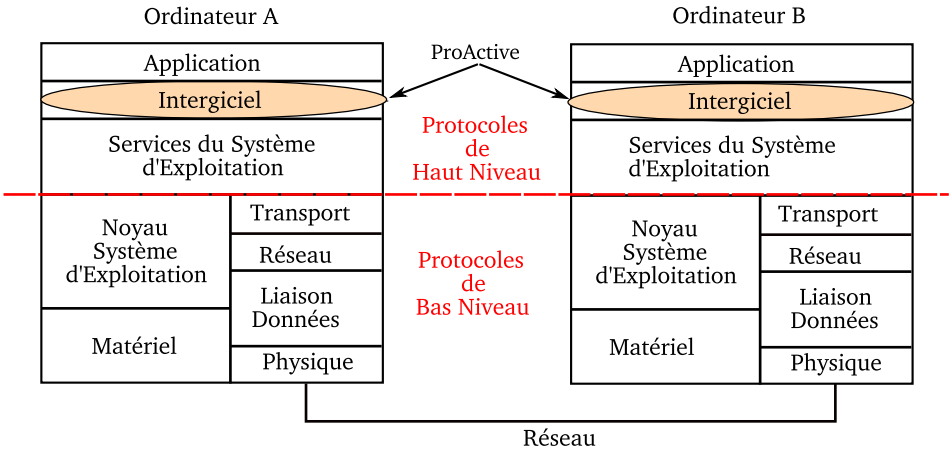


FIG. 2.3 – Couche des protocoles

moniteur de référence afin d'accéder à l'objet sans passer par le mécanisme de contrôle d'accès. Quand il n'est pas possible de contourner le moniteur de référence, on est en présence de la propriété de *médiation complète*. Les notions de sujet et d'objet ne sont pas statiques, le sujet d'une

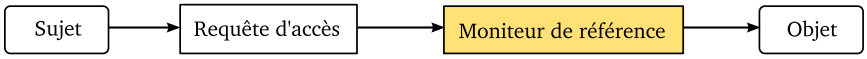


FIG. 2.4 – Moniteur de référence

opération peut devenir l'objet d'une autre opération. Dans les systèmes répartis, une partie du moniteur est composée de protocoles de sécurité permettant entre autres choses de s'assurer, aux moyens de protocoles cryptographiques, l'identité du sujet effectuant l'action sur un objet.

La *matrice d'accès*, le modèle de sécurité proposé par Butler W. Lampson [68], s'appuie sur cette notion de moniteur de référence dans le but d'abstraire et de formaliser les propriétés des systèmes de contrôle d'accès existants. À chaque instant, la matrice d'accès permet d'obtenir les droits attribués à chaque sujet sur chaque objet. Un exemple de matrice d'accès représentant les droits sous `Unix` est présentée par la figure 2.5. Les sujets correspondent aux utilisateurs ou aux processus appartenant aux utilisateurs, les objets correspondent aux fichiers ou aux répertoires. Le contenu de la case de la matrice d'accès située à l'intersection de la colonne o_y et de la ligne s_x

		Objets			
		O1	O2	O3	O4
Sujets	S1	r,w		r	
	S2		x		
	S3	r	r		w

FIG. 2.5 – Matrice d'accès appliquée à la gestion des droits sous unix

représente le *mode d'accès* du sujet s_x à l'objet o_y, il décrit quelles sont les opérations que s_x peut effectuer sur o_y. Lorsque la case est vide, le sujet s_x ne possède aucun droit d'accès sur l'objet o_y. Dans la pratique, les matrices d'accès sont très souvent creuses.

Le modèle à matrice d'accès s'implémente selon deux approches classiques du contrôle d'accès : les *capacités* et les *listes de contrôle d'accès (ACL)*.

– La représentation par capacités [72] correspond à une lecture des droits d'accès de la matrice par ligne. Une capacité correspond à une preuve irréfutable que celui qui la détient est autorisé à effectuer l'action désignée par la capacité sur l'objet donné. Un sujet peut contenir plusieurs capacités. Cette approche permet au sujet de connaître l'ensemble des objets sur lesquels il peut effectuer une action.

– La représentation par ACL correspond à une lecture des droits d'accès de la matrice par colonne. À chaque objet est associée la liste des droits d'un sujet donné. Lors de chaque opération, le mécanisme de contrôle d'accès vérifie que le sujet correspondant possède des droits conformes à l'opération demandée.

Un autre principe à respecter lors de la définition des autorisations est le principe du *moindre privilège*. Un objet ne doit disposer que des droits qui lui sont strictement nécessaires pour réaliser les tâches qui lui incombent.

2.3.1 Contrôle d'accès distribué

Les *stubs* ou *proxies* se sont imposés comme la solution standard dans la construction de systèmes distribués. Leur rôle est de représenter localement un objet distant et masquer de manière transparente l'éloignement physique de l'objet qu'il représente.

Le modèle des *proxies restreints* [92] proposé par Clifford Neuman introduit un mécanisme d'autorisation pour les applications distribuées. Initialement, si un principal A possède un proxy vers un service et qu'il le passe à un principal B alors le principal B possédera les mêmes autorisations que le principal A. Un proxy restreint est un proxy sur lequel des restrictions d'utilisations ont été définies. Il doit être possible pour un serveur de vérifier que les restrictions associées au proxy sont valides avant d'autoriser une opération reçue au travers de celui-ci. Une approche similaire a été proposée dans *Decentralized Jiny Security* [37].

Dans [21], les auteurs présentent une autre approche dans l'utilisation de proxies sécurisés au sein de réseaux de dispositifs mobiles. Ces dispositifs peuvent être soit logiques (un programme), soit matériels (un capteur). Le modèle de sécurité suppose le partage entre l'appareil et le proxy d'une clé symétrique permettant le chiffrement de leurs communications. La partage de la clé est effectué lors de l'initialisation du dispositif.

1. Le proxy et l'utilisateur s'authentifient mutuellement et initient une communication sécurisée.
2. L'utilisateur envoie sa requête au proxy.
3. Le proxy vérifie dans la liste d'accès si l'utilisateur est autorisé à effectuer cette requête. Si l'autorisation est accordée, la requête est émise, dans le cas contraire, un message d'erreur est expédié à l'utilisateur.
4. L'appareil effectue l'action et renvoie une réponse au proxy.
5. Le proxy transfère la réponse à l'utilisateur.

Les proxies présentés dans ces travaux peuvent être qualifiés de *proxies intelligents*. En effet, ces proxies ne se limitent pas à transférer les communications entrantes et sortantes vers l'objet qu'ils représentent mais possèdent des fonctionnalités supplémentaires comme la possibilité de chiffrer les communications avant de les envoyer à l'objet destinataire. Un système de gestion de droit peut aussi être implanté à l'aide de proxies. L'obtention d'un service donc d'un proxy par une entité tierce pouvant être limité dans le temps, il doit être possible d'imposer une durée de validité à un proxy donné.

Cependant, introduire des mécanismes de sécurité au sein des proxies possède un effet de bord, il faut pouvoir contrôler le proxy afin de vérifier les informations qu'il fournit et principalement qu'il est bien un proxy vers celui qu'il prétend représenter.

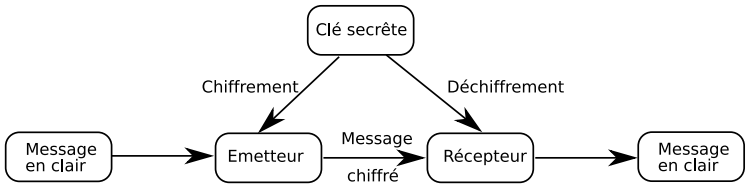


FIG. 2.6 – Chiffrement et déchiffrement avec une clé (Algorithme symétrique)

2.4 Introduction à la cryptographie

La cryptologie, étymologiquement la *science du secret*, regroupe la *cryptographie* et la *cryptalanyse*.

La cryptologie naît il y a environ 4000 ans avec l'apparition de l'écriture. Elle est longtemps considérée comme un ensemble de recettes ou techniques artisanales permettant de rendre un message illisible aux yeux des individus n'ayant pas la connaissance du secret nécessaire pour retrouver le message clair original.

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. Ces algorithmes se classifient en deux grandes familles :

Les algorithmes à clé secrète (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 celle-ci. Le schéma 2.6 illustre le 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 message. Le plus connu des cryptosystèmes à clé symétrique est sans doute *Data Encryption Standard (DES)*[94] introduit en 1977 avec une taille de clé de 56 bits ainsi que son successeur *Advanced Encryption Standard (AES)*[116] offrant la possibilité d'utiliser des clés de 128, 192 ou 256 bits.

Algorithme à clé publique : Les algorithmes à clé publique (également appelé algorithme asymétrique) 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 2.7 illustre le principe de chiffrement à base d'algorithme à clé publique. Les algorithmes à clé publique reposent tous sur des problèmes mathématiques qui ne peuvent, à l'heure actuelle, être résolus en temps polynomial (factorisation de nombres premiers, problème du logarithme discret,...). Par conséquent, ils sont beaucoup plus lents que les algorithmes symétriques. Le cryptosystème le plus couramment rencontré est RSA [104], dû à Ron Rivest, Adi Shamir et Len Adleman.

Protocole hybride

Dans certains 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

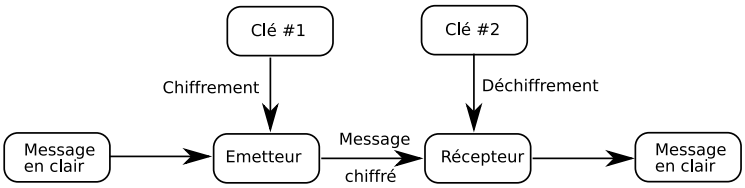


FIG. 2.7 – Chiffrement et déchiffrement avec deux clés (Algorithme asymétrique)

seul l'établissement de la clé de session sera coûteux en temps. En 1976, Whitfield Diffie et Martin Hellman proposent un algorithme hybride connu sous le nom de ses inventeurs *Diffie-Hellman* [32]. Il permet à deux parties de se mettre d'accord sur une clé secrète en communiquant à travers un canal de communication non sécurisé. Cet algorithme est basé sur les travaux de Ralph Merckle [86] concernant l'échange de clés entre deux protagonistes. Une fois cette clé secrète connue par les deux protagonistes, ils peuvent l'utiliser et chiffrer leurs communications en utilisant les techniques cryptographiques usuelles.

Il faut pourtant attribuer la paternité de cet algorithme à Malcolm Willamson qui le décrouvrit quelques années avant alors qu'il était employé par GCHQ, la *British cryptography agency*. L'algorithme n'avait pu être diffusé car il était alors considéré comme un secret militaire.

Signature numérique

Un des plus gros apports 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. Le message chiffré obtenu fait alors office de signature. L'émetteur envoie à la fois le message en clair et sa signature au destinataire. 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).

Fonctions de hachage

Le principe de signature exposé dans le paragraphe précédent présente l'inconvénient d'être coûteux en temps de calcul et en ressources. La signature obtenue est en effet relativement longue à calculer car les algorithmes de cryptographie à clés publiques utilisent des fonctions mathématiques complexes. Les fonctions de hachage apportent la solution à ce problème. Il s'agit d'une fonction mathématique à sens unique qui convertit une chaîne de caractères de longueur quelconque en une chaîne de caractères de taille fixe (souvent de taille inférieure). Le résultat d'une telle fonction est appelé *empreinte*, *somme de contrôle* ou *condensé*. La solution consiste donc à calculer dans un premier temps l'empreinte du message, puis de ne signer uniquement que cette empreinte. Ce procédé est beaucoup plus rapide puisque l'empreinte est de taille beaucoup plus réduite. L'opération de vérification de signature consiste alors à déchiffrer l'empreinte jointe au message, puis de recalculer l'empreinte du message original et de la comparer avec celle qui vient d'être déchiffrée.

Une fonction de hachage doit être résistante aux collisions, c'est-à-dire que deux messages distincts doivent avoir très peu de chances de produire la même signature. On considère l'utilisation d'une fonction de hachage en cryptographie si les conditions suivantes sont remplies :

- il est techniquement difficile (sur le plan matériel ou algorithmique) de trouver le contenu du message à partir de la signature (attaque sur la première préimage) ;
- à partir d'un message donné et de sa signature, il est très difficile de générer un autre message qui donne la même signature (attaque sur la seconde préimage) ;
- il est très difficile de trouver deux messages aléatoires qui donnent la même signature (résistance aux collisions).

Les algorithmes *Secure Hash Algorithm 1* (SHA-1) et *Message-Digest algorithm 5* (MD5), plus ancien et moins sûr, sont des fonctions de hachage utilisées fréquemment.

Cryptanalyse

La *cryptanalyse* s'oppose, en quelque sorte, à la cryptographie. En effet, si déchiffrer consiste à retrouver le clair au moyen d'une clé, *cryptanalyser* c'est tenter se passer de cette dernière. Un algorithme est considéré comme *cassé* lorsqu'une attaque permet de retrouver la clé en effectuant moins d'opérations que via une attaque par force brute. L'algorithme ainsi cassé ne devient pas inutile pour autant, mais son degré de sécurité, c'est-à-dire le nombre moyen d'opérations nécessaires pour le déchiffrer, s'affaiblit.

Une attaque est souvent caractérisée par les données qu'elle nécessite :

- *L'attaque sur texte chiffré seul* (*ciphertext-only*) : le cryptanalyste possède des exemplaires chiffrés des messages, il peut que faire des hypothèses sur les messages originaux.
- *L'attaque à texte clair connu* (*known-plaintext attack*) : le cryptanalyse possède des messages ou des parties de messages en clair ainsi que les versions chiffrées. La cryptanalyse linéaire fait partie de cette catégorie.
- *L'attaque à texte clair choisi* (*chosen-plaintext attack*) : le cryptanalyste possède des messages en clair et les versions chiffrées, il peut également générer ses propres paires de messages. La cryptanalyse différentielle est un exemple d'attaque à texte clair choisi.

Il existe de nombreuses catégories d'attaques possibles suivant la méthode de chiffrement utilisée. Nous présentons ici une liste non exhaustive des principales attaques contre les cryptosystèmes à clés publiques et/ou privées.

- *L'attaque par force brute*
  Il s'agit de tester, une à une, toutes les combinaisons possibles. Dans le cas des cryptosystèmes utilisant des clés symétriques, le nombre de possibilités (si la clé est aléatoire) à explorer est de l'ordre de 2^n où n est la longueur de la clé en bits.
- *la cryptanalyse linéaire*
  La cryptanalyse linéaire, due à Mitsuru Matsui [79], consiste à faire une approximation linéaire de la structure interne de la méthode de chiffrement. Elle remonte à 1993 et s'avère être l'attaque la plus efficace sur DES. Les algorithmes plus récents sont insensibles à cette attaque.
- *La cryptanalyse différentielle*
  La cryptanalyse différentielle est une analyse statistique des changements dans la structure de la méthode de chiffrement après avoir légèrement modifié les entrées. Avec un très grand nombre de perturbations, il est possible d'extraire la clé. Cette attaque est due à Eli Biham et Adi Shamir [17]. On sait maintenant que les concepteurs de DES connaissaient une variante de cette attaque nommée *attaque-T*. Les algorithmes récents (AES, IDEA, etc.) sont conçus pour résister à ce type d'attaque. Les attaques différentielles sont aussi possibles sur les fonctions de hachage, moyennant des modifications dans la conduite de l'attaque. Une telle attaque a été menée contre MD5 [34, 15].

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 robustesse des algorithmes asymétriques est cependant assez délicate, étant donné qu'ils reposent sur des problèmes résolus de manière inefficace à l'heure actuelle. Par exemple, on suppose actuellement qu'il n'existe pas d'algorithme

ayant une complexité polynomiale en temps qui donneraient les facteurs premiers d'un nombre quelconque. Si cette conjecture s'avérait fausse, cela rendrait les cryptosystèmes à clé publique non sûrs. Il se peut également que les découvertes permettent de les valider définitivement en prouvant l'existence de fonctions à sens unique.

Dans [71], 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 compte-tenu de l'évolution de la puissance de calcul des ordinateurs. Leur modèle permet de déterminer la marge de sécurité de la longueur d'une clé en fonction du type de cryptosystème et des quatre paramètres suivants :

- La durée de vie de la clé, c'est à dire le temps de couverture en années nécessaire à la protection de l'information,
- La marge de sécurité qui situe l'infaisabilité de réussite d'une attaque,
- Les évolutions de performance du matériel, la loi de Moore prévoit un doublement de la puissance des processeurs et un doublement de la mémoire disponible tous les 18 mois,
- la cryptanalyse, au même titre que le matériel l'avancement des méthodes de cryptanalyse diminue la marge de sécurité.

Les hypothèses suivantes sont formulées :

1. La puissance de calcul dont disposent les attaquants suit la loi de Moore (i.e double tous les 18 mois).
2. le budget dont disposent les attaquants double tous les dix ans.
3. les découvertes dans le domaine de la cryptanalyse et leurs impacts n'auront pas d'effets importants sur les cryptosystèmes symétriques. Pour les cryptosystèmes asymétriques, il est supposé que les effets de la cryptanalyse de ces cryptosystèmes suivra également la loi de Moore.

La tableau 2.1 présente les tailles de clés que devrait utiliser une application commerciale dont les besoins en terme de sécurité requièrent une protection de la confidentialité et de l'intégrité des données pendant 20 ans. Il s'agit du niveau de sécurité que pouvait offrir le cryptosystème DES en 1982 pour une taille de clé de 56 bits ou RSA pour une taille de clé de 417 bits.

Année	Taille des clés symétriques (bits)	Taille des clés asymétriques (RSA) (bits)
2000	70	952
2002	72	1028
2005	74	1149
2010	78	1369
2040	101	3214
2050	109	4047

TAB. 2.1 – Niveau de confiance d'un algorithme suivant la taille des clés utilisées

2.5 Les infrastuctures à clé publique

Une infrastructure à clé publique (PKI) suppose l'utilisation d'un algorithme à clé publique. Toute la difficulté des mécanismes d'autorisation au sein d'une PKI repose sur la problématique d'être capable d'associer une identité à une clé publique de manière sûre. L'entité à identifier est nommée un *principal*. L'approche standard est de réunir la clé publique et des informations permettant de s'assurer de l'identité du porteur dans un document précis : le certificat. La validité des informations présentes dans le certificat est certifiée par une autorité de certification qui signe numériquement ces informations. La confiance que peut avoir un tiers envers un certificat dépend directement de la confiance qu'il accorde à l'autorité de certification qui a signé le certificat.

L'interconnexion des systèmes rend inévitable le besoin de communication entre les utilisateurs de PKI différentes. Pour résoudre ce problème, des PKI peuvent décider d'une certification

croisée. Chaque autorité de certification émet un certificat pour la clé publique de certification de son homologue. Ainsi, les utilisateurs finaux peuvent établir un chemin de confiance pour les certificats des deux PKI (figure 2.8). Il est également possible à une autorité de certification de déléguer son pouvoir de signer des certificats, on assiste alors à la création de chaînes de certificats.

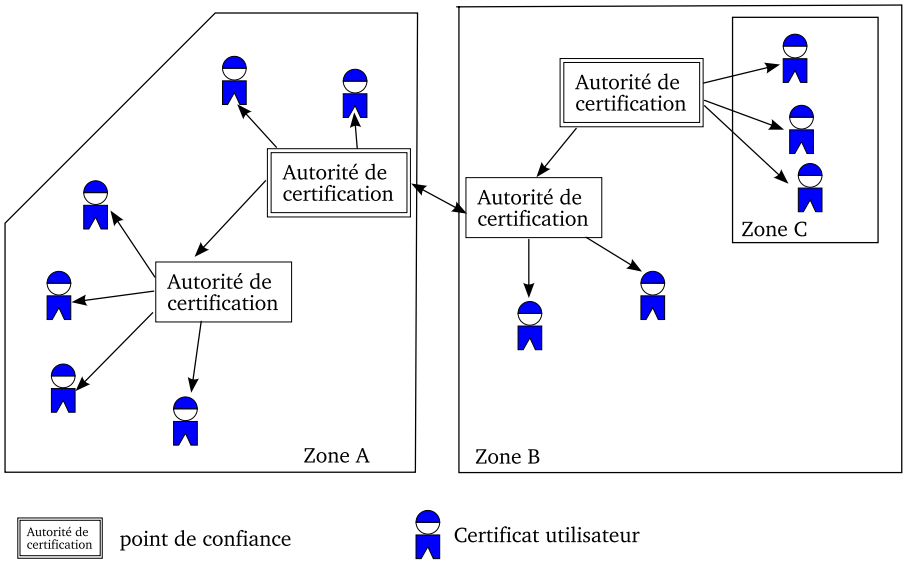


FIG. 2.8 – Certifications hiérarchiques et croisées

2.5.1 X.509

En 1998, l'ITU-T (Organisme international définissant les normes en télécommunication) a présenté la recommandation X.509 comme membre des recommandations de la série X.500 qui définissent un service d'annuaire. Appliqué au modèle des infrastructures à clé publique, l'annuaire représente un serveur qui maintient une base de données des certificats créés. X.509 a été proposé comme infrastructure à clé publique utilisant des certificats et des signatures numériques pour gérer la distribution des clés publiques.

X.509 a un fonctionnement fortement centralisé (figure 2.9), il repose sur l'utilisation d'un ensemble restreint de tierces parties de confiance (autorité de certification) pour fournir la notion de confiance quant à la distribution des clés publiques aux utilisateurs. Seules les autorités de certification peuvent créer des certificats et les signer. Chaque principal possède la clé publique de l'autorité de certification et s'en sert pour vérifier si un autre certificat est valide. Le principal, par définition, fait confiance à l'autorité de certification qui a signé son certificat. L'autorité de certification est la racine de la hiérarchie de confiance d'un annuaire X.500.

La norme X.509 identifie un principal en utilisant des informations telles que le nom, l'adresse email, la ville, etc. Le champ sujet d'un certificat X.509 doit contenir l'identité exacte d'un principal. L'annuaire X.500 exprime cette idée sous le terme de *distinguished name*. Un *distinguished name* est un nom unique qu'un utilisateur pourra utiliser quand il voudra se référer à une entité. Étant donné qu'il s'agit d'un service d'annuaire, l'unicité des noms est importante, la relation entre un principal et son certificat étant une bijection.

Les problèmes inhérents à une architecture à clé publique de type X.509 sont l'obtention de la clé publique de l'autorité de certification, l'obligation d'utiliser des noms (DN) uniques, l'architecture globale construite suite au chaînage de certificats des autorités de certification.

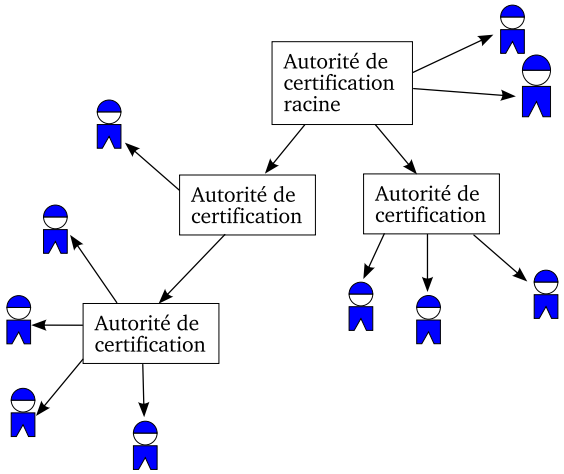


FIG. 2.9 – Une PKI selon X509

2.5.2 Pretty Good Privacy

PGP [126], inventé par Phil Zimmerman, est une autre solution d'authentification à base de certificats. Par opposition à l'architecture X.509 dont la mise en œuvre requiert une forte centralisation et hiérarchisation de ses divers composants, l'architecture de PGP est une architecture complètement décentralisée. Cette décentralisation intervient au niveau de l'autorité de certificat : un principal est lui-même une autorité de certification ; il peut générer et signer ses propres certificats, il lui est également possible de signer les certificats d'autres principaux. Le niveau de confiance d'un certificat est associé directement au niveau de confiance accordé à l'autorité de certification qui l'a généré. Pour X.509, c'est une relation binaire, on fait confiance ou non. PGP revisite cette notion de confiance, un principal pouvant accorder différents niveaux de confiance dans le certificat d'un autre principal. Ce modèle de confiance de PGP stipule que le niveau de confiance accordé à un certificat dépendra du niveau de confiance accordé à tous les signataires du certificat. À chaque nouveau certificat découvert, l'utilisateur va pouvoir calculer le niveau de confiance du certificat en fonction des signataires mais il peut aussi choisir d'attribuer au certificat un niveau de confiance personnalisé. Aussi l'utilisateur d'un système basé sur PGP doit maintenir une base de données associant chaque certificat avec un niveau de confiance.

Cette solution décentralisée et simple a rendu le mécanisme PGP populaire auprès des personnes désirant obtenir des mécanismes de sécurité basés sur une architecture à clé publique sans pour autant devoir subir la difficile mise en place d'une infrastructure à clé publique du genre de X.509. Le cryptosystème PGP est surtout utilisé comme outil de chiffrement pour la messagerie électronique, chaque email pouvant être signé et/ou chiffré numériquement.

2.5.3 SPKI/SDSI

La lourdeur d'une infrastructure à clé publique basée sur X.509 a amené la communauté à chercher d'autres solutions de sécurité basées sur une infrastructure à clé publique. SPKI (Simple Public Key Infrastucture) [35, 36] est le fruit de ces recherches. Il propose un système de sécurité distribué simple à mettre en œuvre.

Un des premiers principes de SPKI est de revenir sur la définition du certificat qui, dans X.509, associe une personne physique à une clé publique. SPKI lève cette limitation en permettant d'attribuer une paire de clés publique/privée à n'importe quelle entité. Bien évidemment la notion de propriétaire (personne, organisation, ordinateur) d'une clé est possible et autorisée mais elle n'est pas nécessaire. Le propriétaire contrôle la clé publique et on peut la voir comme

un *proxy* pour un individu.

Le deuxième principe est de considérer toutes les clés comme égales. Contrairement à X.509 où seules les autorités de certification peuvent certifier un certificat, tout principal (certificat) est capable de générer des certificats donc des principaux (certificats). De plus, l'infrastructure de sécurité ne repose pas sur une architecture centralisée ; cette approche permet le déploiement facile et rapide d'architectures sécurisés autonomes.

Chaque principal peut assigner un nom choisi arbitrairement à un principal, l'unicité globale des noms n'est plus requise et ils peuvent être définis de manière à être compréhensible par des humains. Les noms assignés dans un principal ne restent valides que dans le contexte de ce principal. Ainsi des noms génériques comme "maman" ou "papa" peuvent être utilisés par plusieurs principaux sans pour autant faire le lien vers les mêmes principaux.

SDSI (Simple Distributed Security Infrastructure) [90, 80] désigne le mécanisme de nommage existant au sein de SPKI. Les politiques de sécurité sont conçues pour s'exprimer au sein de listes de contrôle d'accès. Afin de faciliter leur gestion, il est possible de regrouper les principaux au sein de groupes et d'utiliser ces groupes au sein des politiques de sécurité ou d'utiliser les noms symboliques définis au sein des certificats.

Cependant le modèle présuppose qu'un principal soit à tout moment accessible pour fournir un ensemble de services comme l'accès aux certificats générés par ce principal et l'accès aux autres objets appartenant au principal qui ont pu générer des certificats.

2.6 Les politiques de sécurité

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.

L'une des applications de la sécurité informatique est celle du contrôle d'accès aux informations jugées sensibles. Afin de définir quels sont les sujets autorisés à accéder à une information sensible, ou encore quelles sont les opérations autorisées et celles qui sont interdites, on établit une *politique de sécurité*. La notion de politique de sécurité définit à la fois un ensemble de propriétés de sécurité qui doivent être satisfaites par le système et un schéma d'autorisations qui représente les règles permettant de modifier l'état de protection d'un système.

Une définition générale d'une politique de sécurité peut être la suivante :

Définition 2 Politique de sécurité *: Ensemble des lois, règles et pratiques qui régissent la façon de gérer, protéger et diffuser les informations et autres ressources sensibles au sein d'une organisation [26].*

Nous allons tout d'abord étudier plusieurs types de politiques de sécurité : les politiques de contrôle d'accès discrétionnaire, les politiques de contrôle d'accès obligatoire et les politiques à base de rôles.

Nous pouvons distinguer deux types de sécurité utilisant les politiques de sécurité présentées précédemment, 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 [50]. Il faut établir des règles permettant à ces politiques de coopérer dans le but d'obtenir un système sécurisé optimal.

2.6.1 Politique de contrôle d'accès discrétionnaire

Les politiques de contrôle d'accès discrétionnaire (ou DAC pour *Discretionary Access Control*) sont basées sur les notions de sujets, objets et droits d'accès. Les droits d'accès à chaque information sont manipulés par le propriétaire de l'information. On parle alors de *contrôle d'accès discrétionnaire*. C'est le type de politique de sécurité qui vient le plus naturellement à l'esprit lorsqu'on veut définir des règles d'accès à des objets car on associe directement sur l'objet les politiques de sécurité dont il dépend.

Définition 3 *Contrôle d'accès discrétionnaire*
Le contrôle d'accès est dit discrétionnaire lorsque la méthode de gestion de l'accès aux objets est basée sur l'identité du sujet. Le contrôle est discrétionnaire dans le sens où un sujet possédant un droit d'accès est capable de conférer ce droit à tout autre utilisateur.

Une politique discrétionnaire n'est applicable que dans la mesure où il est possible de faire totalement confiance aux sujets (et utilisateurs) ; une telle politique est par là même vulnérable aux abus de pouvoir provoqués par maladresse ou malveillance. S'il est possible à un sujet d'accéder à certains objets, il est alors possible qu'un programme malveillant s'exécutant sous la même identité accède à des données sensibles et puisse les diffuser à des tierces personnes non autorisées.

Ce type de politique se retrouve dans la plupart des systèmes d'exploitation actuels (famille des Unix, Windows) dans lesquels pour chaque objet (fichiers, répertoires) une liste de contrôle d'accès associe un utilisateur à une liste de droits (lire, écrire, exécuter).

2.6.2 Politique de contrôle d'accès obligatoire

Une politique de contrôle d'accès obligatoire (ou MAC pour *Mandatory Access Control* porte non plus seulement sur l'accès mais également sur le flux d'information contenu dans les objets. A l'opposé des politiques de contrôle d'accès discrétionnaires, les sujets d'une politique de contrôle d'accès obligatoire ne sont plus propriétaires des informations auxquelles ils ont accès. De plus, l'opération permettant la délégation des droits est contrôlée par les règles de la politique. Les sujets n'ont plus de pouvoir sur les informations qu'ils manipulent. Le sujet n'a accès à une information que si le système l'y autorise.

Définition 4 *Contrôle d'accès obligatoire*
Le contrôle d'accès est dit obligatoire lorsque l'accès aux objets est basé sur le niveau de sensibilité de l'information contenue dans les objets. L'autorisation d'accéder à un objet est accordée à un sujet si le niveau d'autorisation de ce sujet est en accord avec le niveau de sensibilité de l'information.

Ces règles diffèrent selon qu'il s'agisse de maintenir des propriétés de confidentialité ou d'intégrité. Concernant les politiques obligatoires pour la confidentialité, les politiques les plus souvent employées sont des politiques *multi-niveaux*. Cette politique repose sur des classes de sécurité des informations et des niveaux d'autorisation (figure 2.10). La définition d'une politique multi-niveaux selon Gasser [51] spécifie que tout objet du système possède une *classe de sécurité* et que tout sujet possède un *niveau d'autorisation*. Le modèle de Bell-LaPadula [13] fournit un exemple d'utilisation de telles règles. Des politiques obligatoires similaires peuvent être créées afin de garantir l'intégrité d'un système. C'est le cas de la politique que propose Biba [16] qui applique pour l'intégrité un modèle similaire à celui proposé par Bell-LaPadula. Clark et Wilson [24] proposent un autre type de politique obligatoire pour l'intégrité. Dans leur modèle, les objets sont de deux classes d'intégrité : Les *Unconstrained Data Items* (UDI ou données non contraintes) et les *Contrained Data Items* (CDI ou données contraintes). Les CDI sont certifiés par des procédures de vérifications d'intégrité (IVP) et ne peuvent être manipulées qu'au travers de procédures de transformation (TP) certifiées. Le système maintient une liste indiquant quelles CDI peuvent être manipulées par une TP et quel sujet peut exécuter une TP.

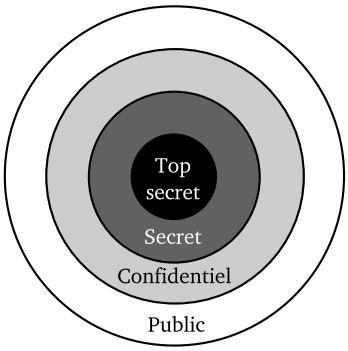


FIG. 2.10 – Confinement des données au sein de classes

Une politique de contrôle d'accès obligatoire n'est applicable que dans la mesure où toutes les parties du système sont sous le contrôle d'une même entité. Les relations d'ordre partiel sont mal adaptées lorsqu'on se trouve en présence d'une gestion décentralisée des niveaux. Par exemple, au sein d'une même entreprise, il est difficile sinon impossible de spécifier si une information venant d'un service est plus secrète ou plus critique que celle provenant d'un autre service sans expliciter de règles de conversion des classes de niveau entre les services. L'interaction de deux systèmes gérés par des entités différentes et dont les niveaux de sensibilité de l'information et les niveaux d'autorisation ne sont pas communs ou tout au moins équivalents peut mener à des failles au niveau de la sécurité. Même si, indépendamment les deux systèmes sont sécurisés, le système global résultant peut contenir des failles. Un système de politique de contrôle d'accès obligatoire peut être utilisé aisément dans une administration centrale. Il n'est cependant pas adapté pour sécuriser les interactions de diverses organisations indépendantes.

2.6.3 Politiques de contrôle d'accès à base de rôles

On constate généralement que dans les grandes organisations ou entreprises, les utilisateurs finaux ne sont pas les propriétaires des documents qu'ils manipulent et pour lesquels ils ont un droit d'accès. L'entreprise est le réel propriétaire de ces informations et des processus qui permettent de les manipuler. Le contrôle d'accès est le plus souvent réalisé sur les rôles qu'assument les utilisateurs au sein de l'entreprise plutôt que sur la propriété des données comme c'est le cas dans le contrôle d'accès discrétionnaire.

Définition 5 *Politique de contrôle d'accès à base de rôle*
Une politique de contrôle d'accès à base de rôle (RBAC pour Role Based Access Control) se base sur la description des fonctions qu'un sujet à le droit d'accomplir au sein d'une organisation pour établir les règles d'accès aux informations [40, 107].

Chaque organisation peut créer pour son propre compte une politique interne basée sur les rôles des divers sujets au sein de l'organisation. Ce type de politique convient particulièrement bien aux entreprises. Il suffit généralement de définir une fois pour toute la politique en regard de tous les rôles existants au sein de l'entreprise. Par la suite, lorsqu'un nouvel employé est recruté, il suffira de lui assigner les rôles qu'il est censé avoir au sein de l'entreprise pour que le système puisse automatiquement cantonner ce nouveau sujet dans les limites autorisées par la politique. Ce type de politique de contrôle d'accès est utilisé au sein des systèmes informatiques de grandes banques européennes [109].

2.6.4 Politiques de sécurité distribuées

L'un des problèmes les plus importants survenant dans les systèmes utilisant une gestion distribuée de la sécurité 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 desquelles peut résulter 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 [55], celui de McCullough [81] ou encore celui de McLean [84].

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* entres 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 [85] 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 [43] propose un modèle de combinaison s'appuyant sur le langage de spécification Z qui est un langage basé sur la théorie des ensembles et la logique du premier ordre (cf. [33]). Hosmer [59] 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.

2.6.5 Bilan

Le problème lié à l'existence de différentes politiques de sécurité dans les systèmes décentralisés 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.

Si on compare toutes ces politiques, on s'aperçoit rapidement qu'on peut les classer en deux catégories. La première est celle contenant les politiques de contrôle d'accès discrétionnaire. La deuxième catégorie regroupant les politiques de contrôle d'accès obligatoire et les politiques à base de rôles. Le facteur différenciant ces deux catégories est la structuration implicite qui est imposée aux utilisateurs de ces divers modèles. Une politique d'accès discrétionnaire ne requiert aucune organisation logique. Contrairement à cela, les membres de la deuxième catégorie imposent l'existence d'un d'ordre partiel permettant la classification des sujets. De plus, il faut une structure externe afin de contenir cet ordre partiel. Le choix d'un modèle de politique de sécurité dépend donc fortement du contexte dans lequel les règles devront pouvoir être exprimées.

2.7 Programmation Réflexive

Cette thèse se propose d'utiliser certains aspects de la programmation réflexive afin de rendre les mécanismes de sécurité orthogonaux au code de l'application. Nous présentons tout d'abord les concepts de métaprogrammation et de programmation réflexive, puis nous étudions comment l'utilisation de la programmation réflexive peut s'intégrer avec des mécanismes de sécurité.

2.7.1 Principes de base

Afin d'exposer les principes de base de la programmation réflexive, nous commençons en introduisant l'idée fondamentale de la métaprogrammation : un programme peut être considéré comme une donnée qui peut faire l'objet d'un traitement de la part d'un autre programme.

La faculté de pouvoir considérer un programme comme une donnée a conduit à la définition des concepts de *métaprogrammation*, *métaprogramme* et *programme de base*.

Définition 6 Métaprogrammation *:*
La métaprogrammation *est la technique de programmation qui consiste à représenter un programme sous la forme d'une donnée sur laquelle un autre programme peut raisonner et agir.*

Définition 7 Métaprogramme *:*
Un programme qui agit sur un autre programme, considéré à ce moment-là comme une donnée pour lui, est appelé métaprogramme*.*

Définition 8 Programme de base *:*
Le programme considéré comme une donnée par le métaprogramme est appelé programme de base*.*

La figure 2.11 (a) représente la relation *raisonne et agit sur* pour un métaprogramme et le programme de base sur lequel il agit. Un interpréteur, un débuggeur ou un analyseur de performances, par exemple, sont des métaprogrammes pour les programmes sur lesquels ils agissent.

Dans le cas particulier où le métaprogramme agit sur lui-même, on est alors en présence d'un système *réflexif* [111] (figure 2.11 (b)). D'après la définition donnée par Pattie Maes [78], un système réflexif est défini par sa faculté de *raisonner et agir sur lui-même*. Une définition plus complète a été donnée par Brian Smith dans [61]

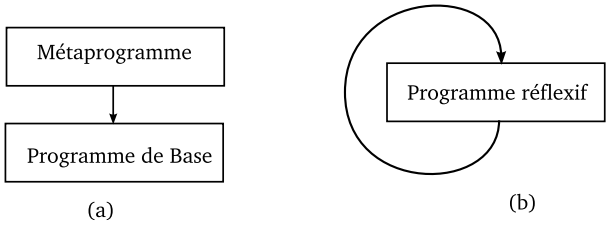


FIG. 2.11 – Relation *Raisonne et agit* entre un programme de base et un métaprogramme

Définition 9 *Réflexion :*
la réflexion est la capacité propre d'une entité à se représenter et agir sur elle-même de la même manière qu'elle représente et agit sur son objet habituel[1].

2.7.2 Interception transparente des appels de méthode

L'interception transparente des appels de méthodes envoyés à des objets réflexifs constitue la partie implicite d'un *protocole à méta objets* (MOP). En effet, ni l'objet qui effectue l'appel, ni l'objet auquel l'appel est destiné ne savent que l'appel est dérouté vers le méta niveau.

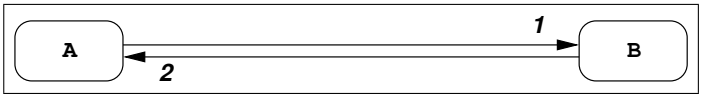


FIG. 2.12 – Transfert d'un appel de méthode de A vers B.

Sur l'exemple présenté dans la figure 2.12, une instance de la classe A appelle une méthode sur une instance de la classe B. Dans cette figure, comme dans les figures suivantes, une flèche dénote un transfert de contrôle depuis une instance *d'une classe* vers une instance *d'une autre classe*. Ainsi, à chaque appel de méthode correspondent deux flèches, une pour l'appel proprement dit et une autre pour le retour du flot d'exécution à l'appelant. Les chiffres qui accompagnent les flèches indiquent l'ordre chronologique des appels successifs. Notre objectif est d'intercepter l'appel de méthode de A vers B afin de transférer le message à un méta objet qui pourra contrôler l'envoi de ce message par A ainsi que les paramètres de sécurité associé mais aussi sa réception par B.

[1]Reflection : an entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter.

Nous présentons ici la technique d'interception de l'appel de méthode par *objet d'interposition*. Cette technique d'interception transparente de l'appel de méthode est basée sur l'utilisation d'un objet d'interposition qui s'interpose entre deux objets de manière à intercepter les transferts de contrôle de l'un vers l'autre. Il s'agit en fait d'une implémentation du design pattern *Proxy* [49]. Le graphe des appels de méthode successifs est présenté sur la figure 2.13 où l'objet d'interposition est appelé `I`. On constate que l'objet d'interposition intercepte l'appel de méthode à la fois au moment de son transfert de `A` vers `B` et aussi lors du retour du flot d'exécution de `B` vers `A`.

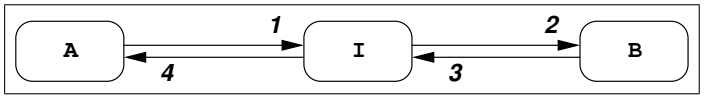


FIG. 2.13 – Utilisation d'un objet d'interposition entre `A` et `B`

L'intérêt d'une approche à objets d'interposition pour la programmation à objets réflexive a été identifié pour la première fois par Geoffrey A. Pascoe [98] pour le langage Smalltalk. La mise en œuvre de cette technique dans un langage à typage dynamique comme Smalltalk est en fait beaucoup plus aisée que dans le cadre d'un langage à typage fort et statique comme Java. En effet, l'implémentation en Smalltalk consiste à créer la classe de l'objet d'interposition de manière à ce qu'elle ne reprenne aucune des méthodes de la classe encapsulée[2]. Les appels de méthode envoyés vers l'objet réflexif `B` mais interceptés par l'objet d'interposition `I` ne trouvent par conséquent aucun sélecteur leur correspondant, et la méthode par défaut `doesNotUnderstand` est appelée. C'est à l'intérieur de cette méthode que le passage au niveau méta est implémenté. Cette technique est d'un usage répandu en Smalltalk et a aussi été utilisée, par exemple, pour implémenter des références transparentes vers des objets distants [14, 82]. Dans le cadre de *ProActive* (Java), l'objet d'interposition est polymorphiquement compatible avec l'objet qu'il représente, ce polymorphisme est assuré par la génération dynamique de l'objet d'interposition comme sous-classe de l'objet cible.

Lorsque nous faisons figurer le méta objet dans la chaîne d'appels (voir figure 2.14), nous constatons que c'est le méta objet lui-même qui se charge d'exécuter l'appel de méthode réifié en transférant l'appel de méthode à l'objet `B` puis récupère le flot d'exécution lors de la sortie de la méthode. C'est aussi le méta objet qui communique le résultat de l'appel de méthode à l'objet d'interposition, qui lui-même le renvoie à l'objet appelant.

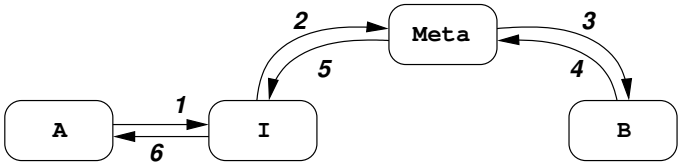


FIG. 2.14 – Transfert de contrôle au méta niveau.

Il est indispensable que la présence du mécanisme d'interception des appels de méthode soit complètement transparente au programme de base. En d'autres termes, la sémantique du programme de base doit rester inchangée si le comportement de niveau méta consiste à ne rien faire. Les conséquences de cette propriété sont multiples car elles affectent différents aspects de l'exécution du programme.

2.7.3 Les protocoles à méta objets et la sécurité

Maintenant que nous avons introduit le mécanisme d'interception transparent des appels de méthode, nous pouvons aborder le thème de la sécurité au sein de programme réflexif. Une étude sur les propriétés de sécurité des protocoles à méta objets est présentée dans la thèse de Julien Vayssière [117].

[2]En Smalltalk, les méthodes sont appelées *sélecteurs*.

Il existe peu de publications abordant le problème de la sécurité dans le contexte de la programmation réflexive. La première mention de ce problème se trouve dans les actes du workshop *Reflection and Metalevel Architectures in Object-Oriented Programming* [61], qui s'est tenu en 1990 dans le cadre de la conférence commune OOPSLA/ECOOP. Le consensus parmi les participants à ce workshop semblait être que les systèmes réflexifs introduisent une menace de sécurité d'un type nouveau, et que "de telles fonctionnalités sont potentiellement dangeureuses si elles ne sont pas contrôlées de manière pertinente"[3]. Cependant, aucune solution, ou ébauche de solution, ne fut proposée pour sécuriser les applications réflexives. Bien entendu, le problème de la sécurité des applications était beaucoup moins crucial il y a douze ans qu'aujourd'hui, et la communauté de la métaprogrammation avait d'autres problèmes plus immédiats à résoudre.

A plusieurs reprises dans la dernière décennie, les concepteurs et implémenteurs de MOPs ont abordé le problème de la sécurité dans leurs travaux, mais la plupart du temps il ne s'agissait que de remarques annexes à leur travail, et non pas d'un élément central dans la conception d'un MOP.

Oliva et Buzato dans [95], par exemple, présentent quelques idées sur les moyens à mettre en œuvre afin de discipliner les interactions entre les métaobjets et le niveau de base dans leur *VM-based runtime MOP Guaraná*. Leur approche repose sur l'utilisation d'objets implémentant le pattern *Factory*. Ces objets encapsulent un ensemble d'opérations que tout objet possédant une référence sur cette *factory* peut utiliser. Ils remplissent donc le rôle de *capability* en termes de sécurité : ils sont une sorte de ticket qui confère à qui le possède certains droits d'accès. Lors de leur construction, les méta objets fournis par Guaraná possèdent un objet de ce type qu'ils peuvent passer à tout autre objet. Cependant, l'implémentation de Guaraná repose sur une version 1.0.6 de Kaffe qui est seulement compatible JDK 1.1, c'est-à-dire qu'elle ne peut pas utiliser l'architecture de sécurité de Java 2. En particulier, cela signifie que Guaraná ne supporte pas les notions de certificats, de clé publiques, etc apparues seulement dans le JDK 1.4.

Welch et Stroud dans [120] présentent également les problèmes de sécurité soulevés par leur transformation-based runtime MOP Kava. Ils introduisent l'idée d'une séparation claire entre les classes qui constituent le noyau du MOP, à qui l'on fait confiance, et les classes de métaniveau développées par des parties tierces et auxquelles on n'accorde pas de confiance *a priori*. Néanmoins, rien n'est dit sur comment protéger les objets de base des objets de niveau méta.

Bien que des solutions à ce problème aient été trouvées dans certains cas particuliers, le consensus à l'heure actuelle dans la communauté des agents mobiles semble être que le problème ne peut pas être résolu dans sa généralité. En tout état de cause, il ne nous semble pas que les techniques mises en œuvre pour résoudre le problème du *malicious host* puissent être appliquées au problème qui nous intéresse car elles ne sont pas suffisamment générales.

D'un autre côté, si nous considérons que les MOPs ne sont, après tout, qu'une manière différente d'implémenter des comportements non-fonctionnels au dessus de comportements fonctionnels, il peut être intéressant de regarder les solutions *ad hoc* existantes et comment elles abordent le problème de la sécurité. Le standard des Enterprise Java Beans [114], par exemple, permet de spécifier de manière déclarative quels sont les comportements non-fonctionnels que l'on souhaite ajouter à un composant, tels que la persistance, la gestion des transactions ou la sécurité. Cependant, les EJBs considèrent que toutes les classes implémentant les comportements non-fonctionnels (donc équivalentes à des classes de niveau méta) sont dignes de confiance. La sécurité s'applique aux interactions entre composants, et non pas aux interactions entre les parties fonctionnelles et non-fonctionnelles d'un composant. L'architecture de sécurité présentée dans cette thèse se base sur une approche similaire en se reposant sur le fait que les classes gérant la sécurité sont implantées au cœur de l'intergiciel et sont donc de confiance.

2.7.4 Les Méta Objets Sécurisés

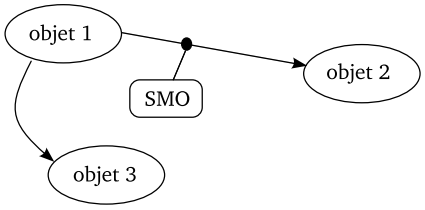
Dans la plupart des modèles basés sur des objets, on peut considérer les références sur les objets comme des capacités. Un objet possédant une référence sur un autre peut potentiellement effectuer toutes les invocations qu'il désire sur l'objet dont il détient la référence. Réciproquement, si un objet n'a pas de référence sur un autre objet, il ne peut pas y accéder. Parallèlement,

[3] *Such powerful capability can be dangerous unless properly controlled.*

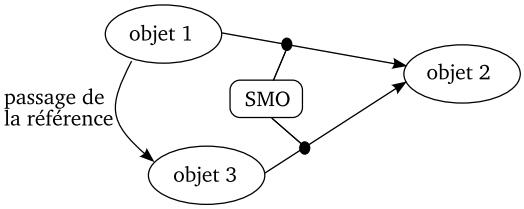
il est difficile pour un objet de connaître les objets qui le référencent et de contrôler la dissémination de ces dernières. La difficulté se rencontre aussi lors de l'utilisation de capacités comme exposé dans [69].

Afin de contrôler l'accès aux objets d'un système, Riechmann et Hauck [103] utilisent un protocole à méta objets permettant d'attacher un objet spécifique, le *security meta object* (SMO), aux références d'un objet (figure 2.15). Cet objet peut fournir les informations du principal lors d'une invocation, garder la trace des références qui sont passées lors d'un appel de méthode et le contrôle d'accès à un objet [102]. Il a l'avantage d'être positionné dans le niveau méta de l'application et ainsi de ne pas être accessible aux objets de l'application. Cette implantation a requis une modification de la machine virtuelle ce qui nuit à la portabilité intrinsèque de Java.

Il est possible d'agréger plusieurs SMO sur une même référence. Afin de contrôler plus finement les références aux objets, les SMO sont configurables et permettent la restriction des droits d'accès, la révocation et l'expiration d'une référence.



(a) état initial du système



(b) passage de la référence

FIG. 2.15 – Une référence sur un objet avec un SMO

Un autre inconvénient de l'approche réside dans sa trop grande granularité qui empêche une vision globale de la politique de sécurité du système. Pour compenser cette limitation, les auteurs ont introduit le concept de *domaine virtuel* [102] qui permet d'imposer une politique globale à un ensemble d'objets, d'empêcher la diffusion des références hors du domaine ou bien d'imposer l'attachement d'un SMO particulier lorsque qu'une référence est passée à un autre domaine. La notion de domaine est une notion récursive permettant d'inclure des domaines dans d'autres domaines afin d'organiser facilement des domaines de politiques de sécurité.

2.8 Sécurisation du code mobile

Le paradigme des agents mobiles est utilisé dans le monde de la programmation distribuée. S'ils sont largement utilisés et malgré leur indéniable supériorité fonctionnelle dans certaines tâches, il faut, du point de vue de la sécurité, bien admettre que cela pose de nouveaux problèmes de sécurité. On ne se retrouve plus avec seulement du *code malicieux* mais aussi des *hôtes malicieux*. L'utilisation d'agents tend à permettre à des utilisateurs de plus en plus nombreux d'accéder à des services offerts par des organisations différentes et peut-être même concurrentes. On est amené à faire cohabiter et collaborer plusieurs applications pouvant ne pas se faire entièrement confiance. Les opérations d'un système à agents mobiles requièrent des services de sécurité

qui s'assurent du respect des politiques de sécurité des différentes parties qui interagissent. Ces politiques ne doivent en aucun cas pouvoir être transgressées par une des parties impliquées que cela soit accidentellement ou intentionnellement. Il existe quatre problèmes de sécurité spécifiques au code mobile :

- La protection d'un hôte contre les agents qu'il exécute,
- la protection d'un agent contre les autres agents d'un même hôte,
- la protection d'un agent contre son hôte,
- la protection d'un agent contre le réseau sous-jacent.

Les différentes attaques possibles concernant le code mobile sont un sujet largement étudié depuis de nombreuses années. Il existe beaucoup de références dans la bilbiographie qui décrivent ces dangers [77, 105, 44, 39].

2.8.1 Protection de l'hôte contre un agent malicieux

Ce type d'attaque consiste à faire exécuter du code mobile malicieux au sein d'un environnement d'execution. C'est l'attaque crainte par tous les fournisseurs de ressources et par tous ceux qui fournissent des services. Ce genre d'attaque a été largement étudié depuis de nombreuses années.

Sandboxing

Le *sandboxing*[4] [54, 83] est présent dans Java depuis la version 1.0 [56] sortie en mai 1995. La possibilité qu'offrait Java de pouvoir créer des *applets* fût en grande partie responsable de son succès. Les *applets* sont des programmes java intégrés au sein de pages web. Il sont chargés et exécutés automatiquement par le navigateur qui accède à la page. Cela pose des problèmes de securité évidents. L'idée principale derrière le sandboxing est la suivante : « Un programme ne peut faire de dégats que si son accès au système d'exploitation sous-jacent n'est pas limité. » En limitant l'accès du programme aux ressources de la machine, on limite les risques. Pour celà on va confiner le code à exécuter dans un environnement dont les accès au système d'exploitation sont contrôlés et limités (figure 2.16). Il faut cependant noter que le code qui s'exécute dans

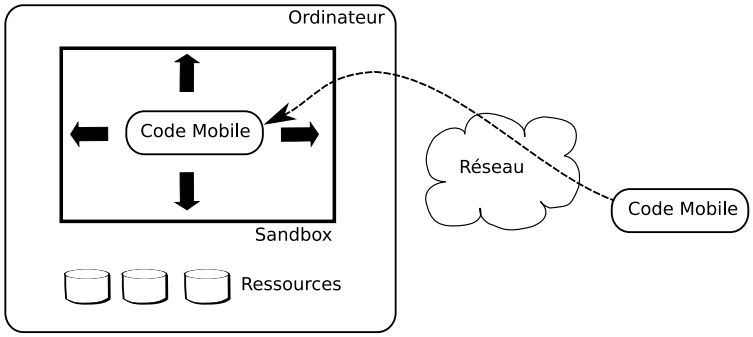


FIG. 2.16 – Sandboxing

une sandbox ne peut avoir que des fonctionalités réduites étant donné qu'il est limité sur les ressources auquel il peut accéder.

Signature du Code

Ce modèle a été utilisé par *Microsoft* au sein de ses *contrôles activeX* et par SUN dans le JDK Java 1.1 sous la dénomination d'*applets signées*. Le créateur du code va signer numériquement ce dernier. Il permet ainsi à toute personne qui le désire de vérifier la signature. Si la signature est valide, on peut être certain de la provenance du code (le fournisseur) et de son intégrité (le code n'aura pas subi de modification) comme présenté dans la figure 2.17. Le code exécuté dans

[4] connu en français sous le nom de « bac à sable »

ce contexte est moins limité en terme de fonctionalités que celui qui s'exécute dans une sandbox. De plus, le code ne sera exécuté que si le système autorise l'exécution du code signé par la clé associée au créateur du programme. Il faut noter que, même si l'intégrité du code pourrait être prouvée, cette solution ne permettrait pas de garantir le comportement de l'application lors de son exécution.

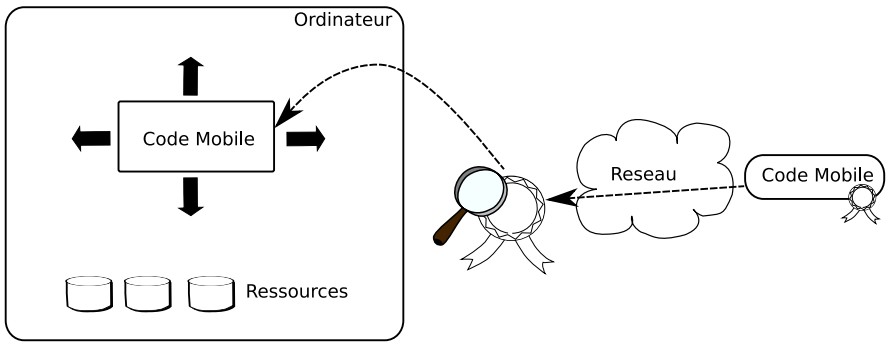


FIG. 2.17 – Signature numérique du code

Ce système est donc dépendant de l'algorithme de hachage générant la signature du code. Si ce dernier n'est pas suffisamment fiable au niveau de l'unicité des empreintes, un attaquant pourrait alors modifier le code de l'agent tout en obtenant une signature identique.

La protection d'un agent contre son hôte est connu dans la littérature sous le nom de *malicious host problem* [58]. Ce problème vise à protéger un programme contre l'interpréteur qui l'exécute. On peut établir un parallèle entre ce problème spécifique au code mobile et l'utilisation d'un protocole à méta objets. Dans les deux cas, l'interpréteur a la possibilité de modifier le code de l'agent s'il le veut vraiment. L'architecture de sécurité présentée dans cette thèse se base sur l'hypothèse que les hôtes sont *de confiance*. Ainsi, si un objet est autorisé à migrer vers un hôte, cela sous-entend que l'hôte est considéré comme de confiance par la personne ayant établi la politique de sécurité de l'objet mobile.

2.9 Conclusion

Dans ce chapitre, nous avons réalisé un état de l'art des différents concepts qui seront utilisés dans cette thèse.

Nous nous sommes attachés dans la première section de ce chapitre à introduire les notions de sécurité dont nous nous servirons tout au long de ce manuscrit. Après avoir défini le vocabulaire couramment employé, nous avons présenté les concepts fondamentaux (problèmes et solutions) de la sécurité informatique en général et plus spécifiquement au sein des systèmes distribués. Nous nous sommes intéressés au contrôle d'accès en exposant le fonctionnement du mécanisme de *moniteur de référence*. Ce mécanisme repose sur deux concepts qui sont, pour le premier, le besoin d'authentification des sujets et pour le deuxième, le besoin d'exprimer des règles d'accès aux objets par les sujets au moyen d'une politique de sécurité. Une solution de contrôle d'accès distribué reposant sur le principe des *proxies restreints* a été présentée.

La section 2.4 nous a permis d'exposer les notions cryptographiques nécessaires à l'introduction des infrastructures à clés publiques présentées dans la section 2.5. L'utilisation d'une infrastructure à clé publique permet de solutionner le problème d'authentification des sujets. De plus, l'utilisation d'un couple de clé publique/privé au sein d'une telle architecture offre la possibilité d'utiliser des protocoles cryptographiques afin de garantir les échanges de données.

La section 2.6 nous a permis, dans un premier temps de définir la notion de politiques de sécurité. Nous avons ensuite énuméré les différents types de politiques de sécurité existants ainsi que leurs spécificités propres. La seconde partie de cette section a mis en évidence les problèmes

posés lors de l'utilisation de politiques de sécurité dans un environnement distribué.

Dans la section 2.7, nous avons abordé le thème de la programmation réflexive ainsi que l'utilisation de protocoles à méta objets et présenté les problèmes posés au niveau de la sécurité dans de tels systèmes. L'interception transparente des appels de méthodes envoyés par et vers des objets réflexifs constitue la base de la partie implicite de notre approche. L'implantation de notre mécanisme de sécurité au sein du méta niveau nous permet, toujours de manière implicite, de mettre en œuvre les objets et les protocoles de sécurité nécessaires sans avoir à modifier le code métier de l'application. Il est ainsi possible de rendre la gestion de la sécurité orthogonale au code métier de l'application.

Finalement nous nous sommes intéressés au problème de sécurisation du code mobile. Notre étude s'est essentiellement portée sur l'analyse des problèmes de sécurité existants dans le cadre des agents mobiles dont l'hôte est considéré comme de confiance.

Chapitre 3

Étude d'intergiciels sécurisés pour le calcul distribué

Il existe déjà de nombreux intergiciels sécurisés. S'ils proposent tous des mécanismes de sécurité basiques permettant l'authentification, l'intégrité et la confidentialité des données, ils n'ont pas la même approche dans la façon de manipuler des objets de sécurité et de faire interagir l'utilisateur, l'application et l'intergiciel.

Nous allons dans ce chapitre présenter les intergiciels que nous avons étudiés. Pour chacun, nous présentons l'architecture générale et le modèle de sécurité utilisé. Afin de permettre une comparaison plus aisée et quand cela nous a été possible, nous présentons l'implantation d'un même exemple pour chaque intergiciel ainsi que les modifications imposées par le mécanisme de sécurité qui en résultent. L'exemple choisi représente l'implantation d'un journal intime accessible à distance. Bien évidemment, seule la personne à qui appartient le journal a le droit d'écrire dedans et de le lire, les communications avec le serveur doivent être confidentielles et intègres. Il s'agit d'un exemple simple, mettant en évidence les mécanismes de sécurité impliqués pour chaque intergiciel ainsi que le niveau de transparence qu'il est possible d'obtenir.

Tous les systèmes présentés ici se basent sur le postulat que les hôtes qui leur permettent de s'exécuter sont de confiance.

3.1 Legion

Développé à l'université de Virginie par l'équipe d'Andrew Grimshaw, Legion [74, 91, 106] est un des premiers projets traitant de grilles de calcul, sa conception remontant à 1994. Dans son principe, Legion se présente comme un méta système d'exploitation à objets orienté grille de calcul. Il étend le système d'exploitation initial en lui rajoutant ses fonctionnalités propres. Cette couche d'abstraction et d'uniformisation permet la réunion d'un ensemble de machines indépendantes et hétérogènes en un seul environnement virtuel partagé par tous les utilisateurs. D'après ses auteurs, Legion est capable de gérer un système contenant des milliers de machines hôtes et autant d'objets, l'ensemble étant interconnecté par des liens réseaux très haut débit.

La philosophie poursuivie par Legion comporte plusieurs aspects dont le point le plus important est le suivant : *Tout est objet*. Tout objet Legion appartient à une *classe* et peut communiquer avec les autres objets par invocation de méthodes. Une classe consiste en une interface, un ensemble d'implantations en divers langages de programmation (C++ et ADA), un générateur ainsi que des informations sur son propriétaire et les besoins en terme de ressources. Les interfaces sont décrites dans un langage d'interface (*Interface Definition Language* ou IDL) indépendant du langage de programmation. Les implantations se divisent en deux parties, les implantations sources contenant le code source de l'objet et les implantations concrètes représentant la version exécutable de l'objet.

Comme n'importe quel système d'exploitation, Legion est construit au dessus de ressources

de bas niveau et fournit les abstractions, les services nécessaires pour exploiter ces ressources sous-jacentes. L'architecture de Legion est une architecture en couches (figure 3.1). Ces diverses couches fournissent les abstractions permettant, en autre chose, la mise en place d'un système de fichiers virtuel existant sur l'ensemble de machines d'un système Legion [123, 124], un système de tolérance aux pannes, et des mécanismes de sécurisation variés [125, 42].

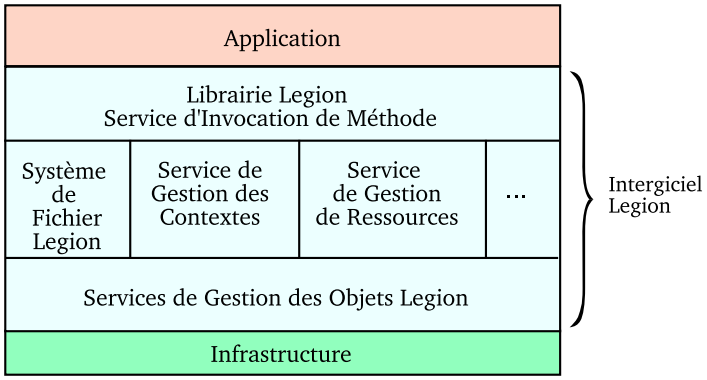


FIG. 3.1 – Architecture de Legion

Tout objet Legion possède un identifiant unique, le *Legion Object Identifier* (LOID). Chaque classe est responsable de l'attribution d'un LOID à ses instances. Le LOID reste cependant un identifiant au niveau du système Legion et difficilement manipulable par un utilisateur. Ce dernier peut créer des contextes au sein desquels il lui est possible de nommer arbitrairement un objet. Ce nom ne sera valide que dans le contexte dans lequel il a été défini. La figure 3.2 présente les deux façons de nommer un objet dans Legion. Notons qu'il est possible de référencer le même objet dans des contextes différents.

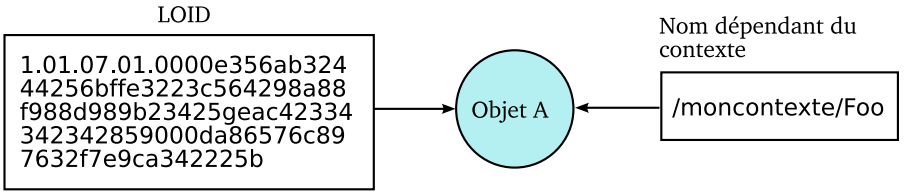


FIG. 3.2 – Les deux façons de nommer des objets dans Legion

Modèle de sécurité

Comme décrit précédemment, Legion est un système orienté objet. C'est pourquoi il n'est pas surprenant que l'unité de base sur laquelle le modèle de sécurité s'applique soit l'objet. Ce principe n'est pas nouveau, il était déjà proposé dans des systèmes comme Hydra (1975) [25] ou Corba. La différence vient de la vision *Tout est objet* que propose Legion. Ainsi, les fichiers et les répertoires sont eux aussi représentés sous forme d'objets sur lesquels les utilisateurs vont pouvoir exprimer des politiques de sécurité.

L'identification des objets se fait grâce aux LOIDs dont une partie contient le certificat X.509 (donc la clé publique) de l'objet. L'authentification des objets se fait au moyen des certificats X.509 en utilisant le mécanisme de chaînage de certificats.

L'autorisation est basée sur un système à capacités (*credential*). Une capacité dresse la liste des autorisations accordées à un objet par son créateur. Il existe deux types de capacités dans Legion :

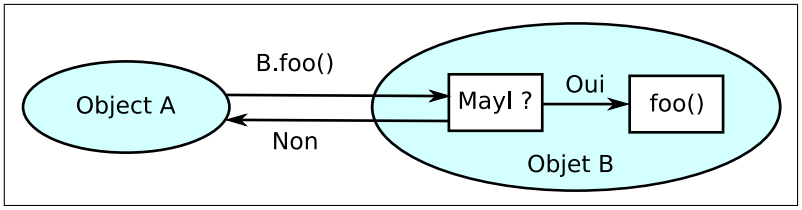


FIG. 3.3 – Modèle de sécurité de Legion

– Le *delegated credential* qui explicite son bénéficiaire et les droits associés.
– Le *bearer credential* qui donne à son porteur tous les droits associés.

Chaque objet Legion possède un service `MayI` qui joue le rôle de moniteur de référence (figure 3.3). L'accès à chaque ressource est contrôlé par une liste de contrôle d'accès qui associe à un LOID une liste de droits. Lors d'une invocation, les mécanismes de transport sous-jacents présentent cet appel au service `MayI` de l'objet. Le service `MayI` de base inclus dans Legion peut être étendu pour supporter d'autres formes de contrôle d'accès.

Un appel de méthode correspond à l'envoi d'un ou plusieurs messages. Chaque message peut être associé à l'un des trois niveaux de sécurité suivant[41] :

– *no security* : le message est envoyé sans modification sur le réseau.
– *protected mode* : seules les capacités associées au message sont chiffrées.
– *private mode* : la totalité du message est chiffrée (capacités et contenu).

En conclusion, le service de sécurité de Legion fournit les fonctionnalités nécessaires à tout système distribué. L'approche consistant à considérer tout le système comme un seul ordinateur élimine d'emblée tous les problèmes de sécurité pouvant survenir lors de l'ajout de nouvelles ressources au sein du système, ils sont traités en amont par les administrateurs. Cette approche limite l'expansion de l'ordinateur virtuel aux seuls sites qui accepteront de modifier leurs systèmes pour accueillir celui de Legion ; on perd la notion de dynamicité des ressources et de systèmes ouverts. La vision unique du système ne permet pas d'adapter les politiques de sécurité selon les localisations des parties communicantes, les niveaux de sécurité requis pour des communications au sein d'un réseau local n'étant pas les mêmes que celles effectuées sur les réseaux publiques.

3.2 Open Grid Service Architecture/Globus

L'initiative *Open Grid Service Architecture* (OGSA) [47] tente de standardiser une architecture pour les grilles de calcul. Elle est principalement supportée par le *Global Grid Forum*. OGSA se base sur les principes des services web afin de définir son architecture. La version 3 du *Globus Toolkit* [46] est une implantation de l'architecture OGSA.

Le Toolkit Globus fournit les services de base nécessaires à la construction d'une grille de calcul. Il est composé de plusieurs modules parmi lesquels :

– La localisation et l'allocation des ressources sont assurées par le *Globus Resource Allocation Manager* (GRAM) qui fournit une interface uniforme aux nombreux protocoles supportés.
– Le service *Monitoring & Discovery System* (MDS) permet d'obtenir des informations sur les ressources et les applications.
– Le service de sécurité *Globus Security Infrastructure* (GSI) que nous allons détailler.

3.2.1 Modèle de sécurité

L'architecture de sécurité de Globus ou *Globus Security Infrastructure (GSI)* [122] est le cœur du système de sécurité de Globus ; c'est sur ce mécanisme que les applications utilisant Globus vont se reposer pour gérer leur sécurité.

Un des buts poursuivis par le Toolkit Globus est de permettre la mise en place d'organisations virtuelles dynamiques [45]. Une telle organisation a pour but de regrouper des groupes

d'utilisateurs, des services et des ressources provenant de plusieurs domaines administratifs différents au sein d'un même ensemble et d'assurer à chaque membre du groupe que n'importe quel autre membre ou ressource de cette même organisation est connu comme une entité de confiance.

L'authentification est basée sur une architecture à clé publique. Chaque entité (utilisateurs, services, ressources) possède un certificat X.509 lui permettant de s'identifier. Les services de sécurité basiques (authentification, chiffrement, intégrité) sont assurés par l'utilisation d'une couche de communication interfacée avec SSL.

L'authentification unique *single sign-on* permet à un utilisateur de s'identifier une seule fois et d'avoir la possibilité d'utiliser les ressources offertes par un système de manière transparente vis-à-vis des mécanismes de sécurité. Cette fonctionnalité est implantée au sein de Globus grâce à l'utilisation de *proxies utilisateurs*. Un proxy utilisateur possède un certificat dérivé du certificat de son créateur via la création d'une chaîne de certificats. Le créateur peut être un utilisateur ou un autre proxy. La création d'un proxy par un utilisateur n'est possible qu'une fois celui-ci identifié et connecté au sein de la plateforme Globus. Les processus créés à partir d'un proxy utilisateur peuvent avoir besoin d'acquérir dynamiquement des ressources. Pour cela, le processus crée la requête de demande d'allocation des ressources et la fait parvenir à son proxy utilisateur. Le proxy, agissant pour l'utilisateur, possède les données nécessaires pour signer la requête et l'exécuter sous l'identité de l'utilisateur qui l'a créé. Le résultat de la requête d'allocation de ressource est signé par le proxy et renvoyé au processus. Si cet algorithme a l'avantage d'être simple à mettre en œuvre, le fait que les requêtes doivent passer obligatoirement par le proxy induit un possible goulot d'étranglement des requêtes au niveau de celui-ci et limite le passage à l'échelle de cette solution.

La figure 3.4 présente les interactions qu'effectue une application distribuée utilisant Globus. Les étapes sont les suivantes : création de processus sur les sites A et B, une communication entre ces deux processus ainsi qu'un accès à des fichiers distants. Toutes ces actions peuvent être sécurisées afin de garantir la confidentialité des échanges.

Dans OGSA, les services web permettent d'exposer des composants logiciels en terme d'interfaces et de lier les méthodes de ces interfaces à des mécanismes de communication spécifiques. Ces informations sont exprimées au sein d'un document utilisant un langage spécifique aux services web, le *Web Service Description Language (WSDL)*. Dès lors, il est possible de spécifier l'utilisation de protocoles de communication capables de gérer la sécurité des messages échangés lors de l'accès à des méthodes de ces composants logiciels. Un exemple d'utilisation de l'infrastructure OGSA et du GSI3 est présenté par la figure 3.5.

Les étapes effectuées pour aboutir à une communication sécurisée entre le client et le service sont les suivantes :

1. L'environnement d'accueil du client récupère et inspecte la politique de sécurité du service afin de déterminer les protocoles, mécanismes et certificats qui sont requis pour soumettre une requête.
2. Si l'environnement d'accueil du client détermine que les certificats ne sont pas déjà associés au client, il contacte le service de conversion de certificats afin d'obtenir les certificats dans le format requis.
3. L'environnement d'accueil contacte le service de traitement des messages afin de formater et manipuler les divers messages d'authentification avant de les expédier au service cible. La délégation de cette tâche à un service permet de libérer le client et son environnement d'accueil de la connaissance des mécanismes de sécurité sous-jacents.
4. Du côté du service cible, l'environnement d'accueil accède à son service de traitement des messages qui peut être le même service que celui utilisé par l'environnement d'accueil du client.
5. Enfin, après l'authentification du client, la requête est dirigée vers le service d'autorisation qui prendra la décision d'accepter ou non la requête.

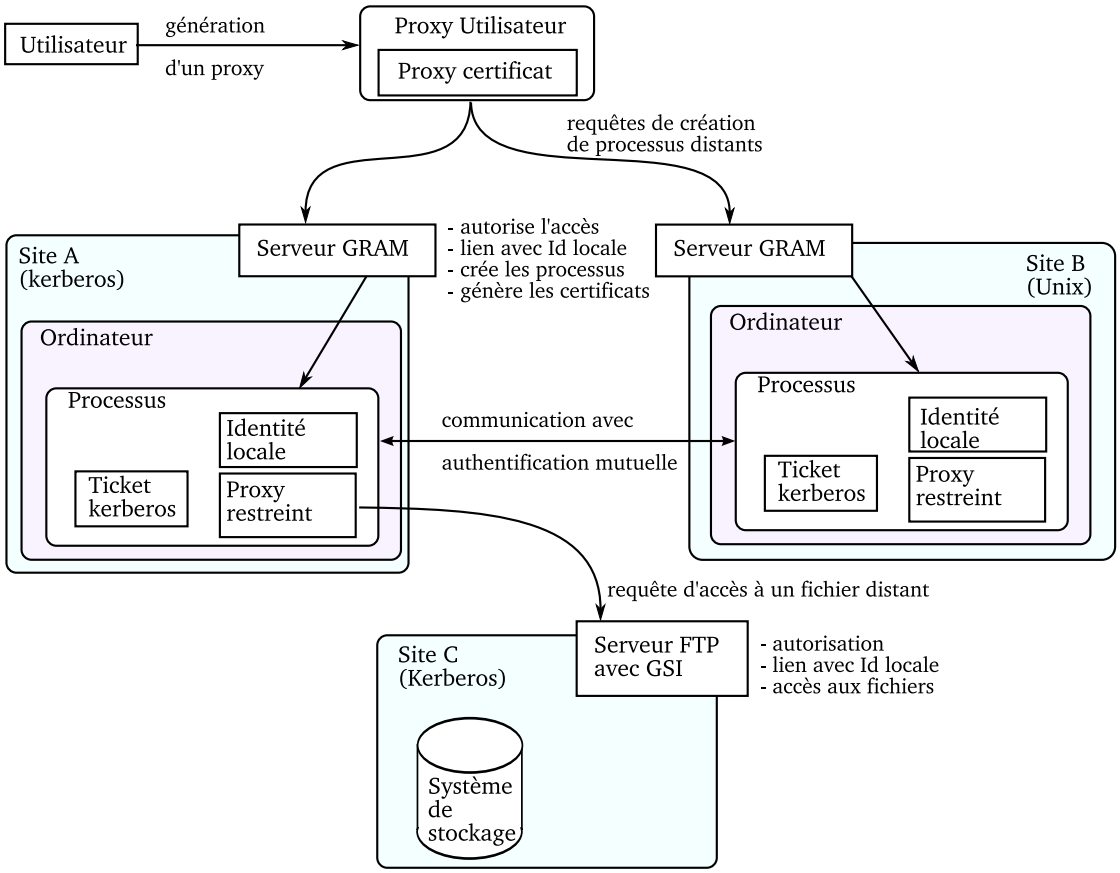


FIG. 3.4 – Globus Security Infrastructure

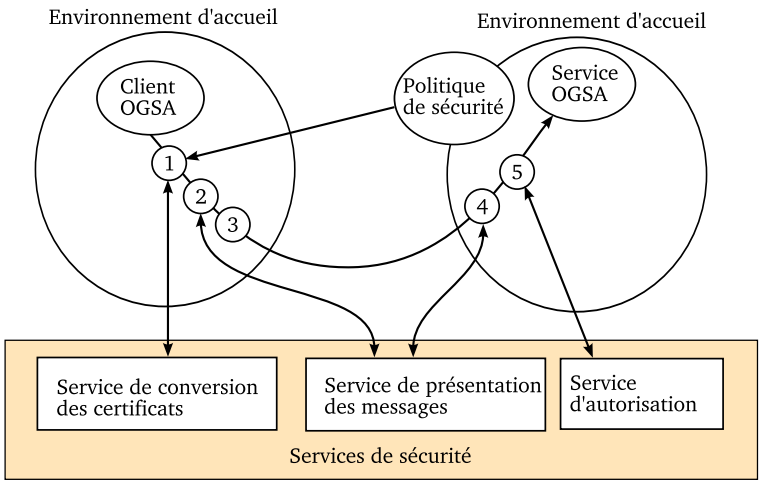


FIG. 3.5 – Modèle de sécurité d'OGSA

Le principal avantage de cette solution est d'extraire du code métier de l'application le code de bas-niveau des mécanismes de sécurité et de ne garder que la gestion de l'identification.

Pendant la rédaction de cette thèse, la version 4 du toolkit *Globus* a été publiée. Cette version apporte notamment le support de la *délégation*. Elle permet d'utiliser un seul *certificat délégué*[1] lors d'invocations multiples de services sur des environnements d'accueils.

3.2.2 Exemple

Pour l'application "Journal Intime", le code gérant l'accès d'un client au serveur est relativement simple. Il suffit dans le code du serveur de vérifier que l'identité du client est bien celle attendue.

Le code du serveur :

```
public class JournalIntimeImpl extends GridServiceImpl {
 ...
public void ajoutEntree(String message) {
  // on recupere l'identite de l'appelant
  String callerIdentity = SecurityManager.getManager().getCaller();

  // on verifie que l'appelant est bien la
  // personne. MON_IDENTITE contient la chaine
  // identifiant l'utilisateur autorise
  if (callerIdentity.equals(MON_IDENTITE) {
     this.ecrireEntree(message);
  }
 }
 ...
}
```

La configuration du service se fait dans un fichier XML annexe utilisé lors de son démarrage. Le fichier ci-après précise que le code serveur doit être exécuté sous l'identité du client et que les messages échangés lors de l'invocation de méthodes doivent être chiffrés et leur intégrité vérifiée.

```
<securityConfig xmlns="http://www.globus.org">
  <method name="ji:ajoutEntree"
      xmlns:ji="http://noadcoco.inria.fr/myservices/journalIntime">
    <run-as>
      <caller-identity/>
    </run-as>
    <auth-method>
      <gsi>
        <protection-level>
          <integrity/>
          <privacy/>
        </protection-level>
      </gsi>
    </auth-method>
  </method>
</securityConfig>
```

Lorsqu'un client veut accéder à un service, il doit tout d'abord récupérer un proxy (stub) vers celui-ci. Ensuite vient l'étape dite de *configuration du stub* qui impose au code client de positionner des variables concernant son authentification ou les conditions de transfert des messages en accord avec la configuration du service distant.

```
// Creation de la requete de creation du service.
JournalIntimeServiceGridLocator journalIntimeGL = new
    JournalIntimeServiceGridLocator();
```

[1] delegated credential

```
// recuperation du stub.
JournalIntimePortType journalIntime = journalIntimeGL.getJournalIntimePort(new URL
    ("http://noadcoco.inria.fr/myservices/journalIntime");

// configuration du stub
// ces deux lignes dependent de la configuration du service que represente
// l'objet local stub
Stub stub = (Stub)journalIntime;
stub._setProperty(Constants.GSI_SEC_CONV, Constants.SIGNATURE);
stub._setProperty(Constants.GSI_SEC_CONV, Constants.ENCRYPTION);

// envoi du message
journalIntime.ajoutEntree("la pensee du jour"));
```

En conclusion, la flexibilité semble être le maître mot de l'approche. Cependant, le support de l'éventail des possibilités offertes par l'utilisation des web services introduit une complexification importante du déploiement de l'application. La transparence des mécanismes de sécurité est gérée de manière asymétrique : une simple gestion des droits d'accès pour le serveur tandis que le client doit configurer des variables du stub qu'il récupère en fonction de la politique exprimée par le serveur. Il n'existe pas à notre connaissance de moyen permettant la configuration automatique des paramètres de sécurité au niveau des stubs.

3.3 CORBA

La spécification CORBA (*Common Object Request Broker Architecture*) fait partie de l'*Object Management Architecture* qui a été définie par l'*Object Group Management* (OMG) [96]. Elle résulte du besoin de normalisation intrinsèque aux applications distribuées. Cette norme permet ainsi de créer une interopérabilité entre toutes les applications distribuées qui la respectent.

Les intergiciels de communication permettent aux objets d'une application de faire, de manière transparente, des appels de méthodes sur d'autres objets disséminés sur le réseau. Cette transparence est rendue possible en déléguant les appels de méthodes distants à un bus logiciel, l'*Object Request Broker* (ORB) ou *courtier d'objets* (figure 3.6). Cet ORB permet d'abstraire les protocoles réseaux du code application et fournit une interface de nommage générique pour les objets. Un ORB sert essentiellement à acheminer les messages d'un objet à l'autre. L'empaquetage/dépaquetage des messages est assuré par le biais d'interfaces d'invocations. Les interfaces des objets CORBA sont standardisées et décrites grâce à un langage commun : l'*Interface Definition Language* (IDL). La localisation des objets est assurée par l'*Interoperable Object References* (IOR).

En complément des fonctions de base de l'ORB, il existe un ensemble de services permettant l'utilisation de services additionnels, les CORBAServices, tels que le nommage, les évènements, la persistance et celui qui nous intéresse tout particulièrement, le service de sécurité.

La spécification du Service de Sécurité de CORBA [9, 29, 18] décrit les diverses caractéristiques du service de sécurité, à savoir l'authentification, la protection des messages, l'autorisation, l'audit et la non-répudiation des messages. Les objectifs de CorbaSec [2] sont de fournir à un ORB les fonctionnalités de sécurité nécessaires sous la forme d'un service. Ses buts sont :

- Assurer que chaque invocation sur un objet est protégée comme requis par la politique de sécurité
- Exiger qu'un contrôle d'accès soit effectué sur chaque invocation
- Empêcher les objets d'une application d'interférer avec ceux d'une autre application.

Plutôt que d'implémenter directement tous ces composants, le service de sécurité présente une interface uniforme aux utilisateurs et utilise des passerelles vers des mécanismes de sécurité déjà existants tels que Kerberos [112], SESAME ou SPKM [1]. L'interface est normalisée suivant la recommandation RFC 2743 GSS-API [75].

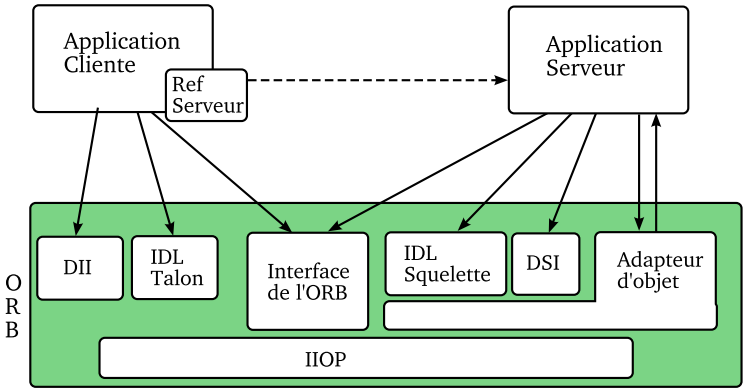


FIG. 3.6 – Architecture CORBA

La première publication des spécifications du service de sécurité de CORBBA date de 1995 et a depuis subit de nombreuses mises à jour afin de réduire les problèmes architecturaux initiaux, notamment ceux portant sur l'interopérabilité et la portabilité. C'est à partir de la version 1.5 de ce document qu'est apparue SSLIOP, une spécification décrivant la communication inter-ORB en utilisant le protocole SSL. Cette spécification a été rajoutée sous la pression des partenaires industriels.

L'architecture du service de sécurité se compose de plusieurs éléments assurant chacun un rôle précis. On retrouve notamment une organisation en domaines de sécurité (*Security Policy Domains*, *Security Domains*). Ils sont constitués d'un ensemble d'objets sur lesquels une politique de sécurité commune sera appliquée. La sécurité de chaque domaine est spécifiée au sein de *domain security policies*. Les domaines sont administrés par une autorité de sécurité *Security Authority* et peuvent être organisés *hiérarchiquement*. Dans ce cas, l'autorité de sécurité délègue la gestion de la sécurité des objets contenues dans le sous domaine à la sous autorité qui contient l'objet en question.

Les domaines de sécurité peuvent aussi être organisés en fédération. Ils ne sont alors plus hiérarchiques mais peuvent être représentés au même niveau. Fédérer deux domaines implique :

1. que chacun des domaines accorde un certain niveau de confiance à l'autre,
2. la combinaison des politiques de sécurité présentes dans chaque domaine.

Il faut, de plus, définir la façon dont les rôles d'un domaine vont correspondre aux rôles de l'autre domaine.

Le rôle de l'ORB sécurisé est de vérifier que la politique de sécurité définie au sein d'un domaine sera appliquée à tous les objets de ce domaine. En pratique la politique de sécurité d'un domaine est contenue au sein d'un serveur de domaine.

Il existe un autre type de domaine : le *Security Technology Domain*. Il regroupe les domaines qui utilisent les mêmes outils pour assurer la sécurité du domaine. Ces outils pouvant être une même API de connexion, la même manière de chiffrer les communications, l'utilisation d'un même algorithme de chiffrement.

Les politiques de sécurité peuvent être imposées à la fois par l'ORB et par l'application elle-même. Les politiques de sécurité systèmes (*system security policy*) sont imposées par l'ORB. Ce type de politiques s'applique à la fois sur toutes les applications, celles qui gèrent leur propre sécurité et celles qui n'ont pas conscience d'utiliser des mécanismes de sécurité. Les politiques de sécurité au niveau applicatif (*Application security policies*) sont imposées par les objets d'une application utilisant explicitement les mécanismes de sécurité. Ces deux types de politiques de sécurité sont regroupés au sein de deux niveaux de sécurité :

– Le *Security Level 1* peut être utilisé quand l'application n'a pas été prévue pour fonctionner avec des notions de sécurité ou qu'elle ne possède que des besoins limités en terme de

sécurité. Ces mécanismes sont implantés au sein de l'ORB. Ils interviennent lors de toutes les invocations sur les objets d'une application. Ils interceptent l'invocation même si l'application n'est liée à aucune notion de sécurité et appliquent sur l'invocation leurs politiques de sécurité. Du côté serveur, il est possible d'obtenir l'identité de l'utilisateur sous laquelle l'application cliente a été lancée.

– Le *Security Level 2* fournit un jeu d'API de sécurité permettant aux applications de gérer explicitement leur politique de sécurité.

Une application gérant explicitement sa sécurité est capable de changer dynamiquement son identité, sa politique de sécurité selon, par exemple, l'identité du serveur distant. Le code java suivant présente les étapes nécessaires à une application pour changer d'identité et contacter un objet serveur distant qui représente un journal intime accessible à distance et dont bien évidemment l'accès doit être restreint au seul propriétaire.

```
// Creation d'un nouveau contexte de securite pour
// stocker les donnees d'authentification
LoginHelper loginHelper = new LoginHelper();
try {

// Indication de l'ID et du mot de passe de l'utilisateur pour
// authentification.
org.omg.SecurityLevel2.Credentials credentials = loginHelper.login(userid,
    password);

// Utilisation des nouvelles donnees d'identification
// pour tous les appels a venir.
loginHelper.setInvocationCredentials(credentials);

// il est possible de recuperer le nom d utilisateur
// associe aux certificats
String username = loginHelper.getUserName(credentials);
System.out.println("Contexte de securite etabli pour : "+username);

// ajout d une entree dans le journal intime
journalIntime.ajoutEntree("un commentaire pour mon journal intime");

} catch (org.omg.SecurityLevel2.LoginFailed e) {
// code de gestion de l echec de l authentification
}
```

La figure 3.7 présente les diverses étapes qui conduisent à l'élaboration de la politique de contrôle d'accès lors de l'invocation d'une méthode à partir d'un objet client sur un objet serveur.

En conclusion, le service de sécurité de CORBA offre un large éventail de mécanismes de sécurité essentiels au sein d'applications distribuées : organisation en domaines de sécurité hiérarchiques, possibilités de fédérer des domaines de sécurité ainsi que la possibilité d'obtenir au moins en partie une intégration implicite des mécanismes de sécurité. Elle se trouve localisée au sein du *niveau de sécurité 1* de l'ORB sécurisé. Ce niveau est dédié à la sécurité d'applications qui n'ont pas été prévues pour fonctionner avec des mécanismes de sécurité. Cependant, ce niveau de sécurité n'est pas configurable par l'utilisateur mais seulement par les administrateurs de l'ORB. Les politiques de sécurité appliquées seront celles des domaines de sécurité. L'utilisateur ne peut exprimer la politique de sécurité qu'il désire pour une application de manière implicite, il est obligé d'utiliser l'API du *niveau de sécurité 2*.

3.4 Les Entreprise JavaBean

Les Entreprise JavaBeans (EJB) [89] sont des composants Java qui peuvent être combinés pour créer des applications distribuées et modulaires. Le standard EJB décrit à la fois une spécification et un ensemble d'interfaces que doivent implanter les composants. Un des principes de conception d'une telle architecture est la séparation des concepts non-fonctionnels *de niveau système* (comme le multi-threading, la persistance, les transactions, la sécurité, etc) de la logique de

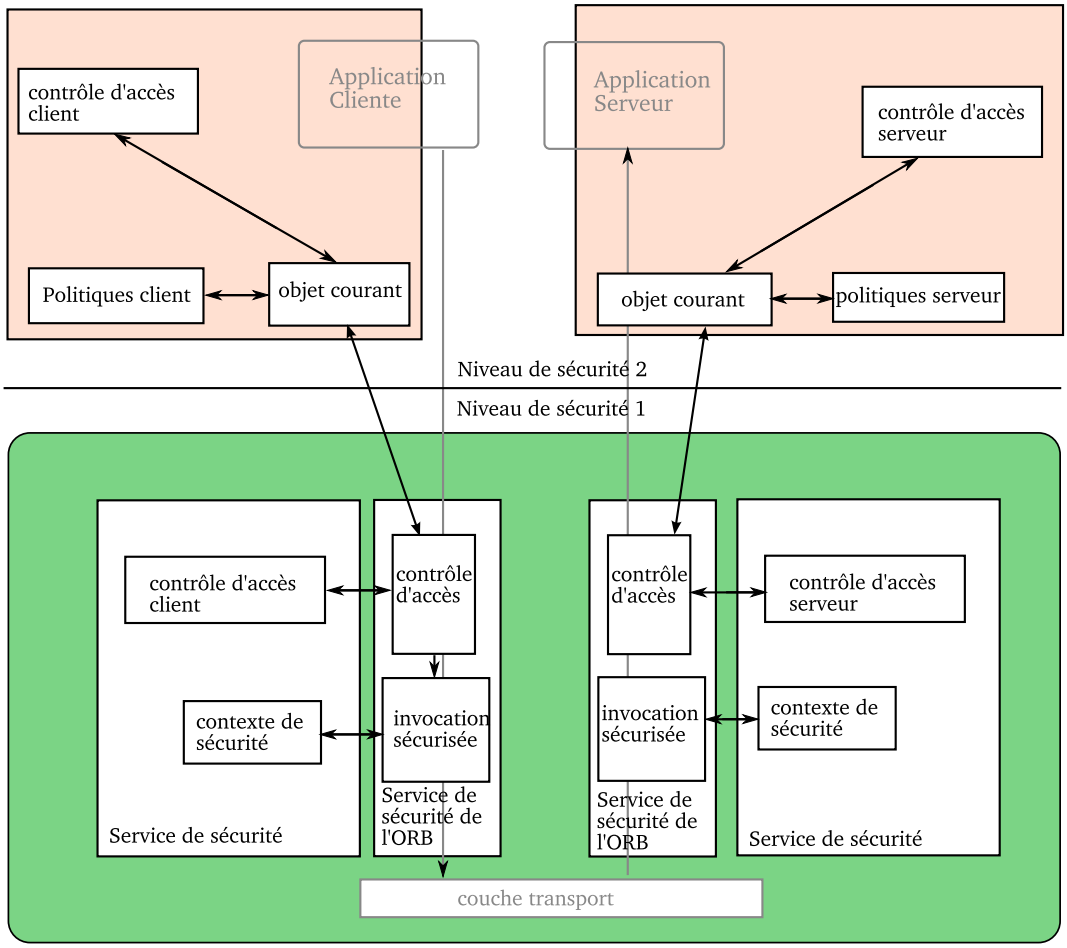


FIG. 3.7 – Architecture d'un ORB sécurisé

l'application (figure 3.8). Des programmes spécifiques nommés conteneurs EJB fournissent une plateforme d'accueil pour les EJB ainsi que les services de niveau système précédemment cités. Ces conteneurs EJB sont eux-mêmes inclus dans des serveurs d'applications qui peuvent accueillir tout un panel de conteneurs différents. De bien des manières, les EJBs ressemblent aux objets RMI, sur lesquels d'ailleurs ils reposent, en exposant leurs fonctionnalités à des clients distants au moyen d'interfaces.

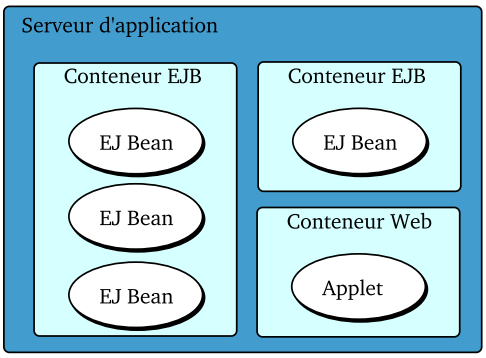


FIG. 3.8 – Exécution d'applications construites avec des EJBs

Les EJBs sont aussi architecturés dans le but de promouvoir leur réutilisation au sein de diverses applications et fournir de la flexibilité dans leurs déploiements et leurs administrations selon les besoins des entreprises. Pour cela, il existe une claire séparation entre les divers acteurs entrant en jeu dans l'élaboration d'une application. Ces acteurs sont organisés en rôle (figure 3.9) :

- les *fournisseurs de Beans* sont les concepteurs des EJBs.
- les *assembleurs d'applications* assemblent les EJBs pour créer une application.
- les *déployeurs d'applications* déploient les applications assemblées au sein d'un environnement de production,
- les *administrateurs systèmes* s'occupent de l'environnement de production, ils veillent à fournir à l'application toutes les ressources dont elle a besoin durant son exécution.

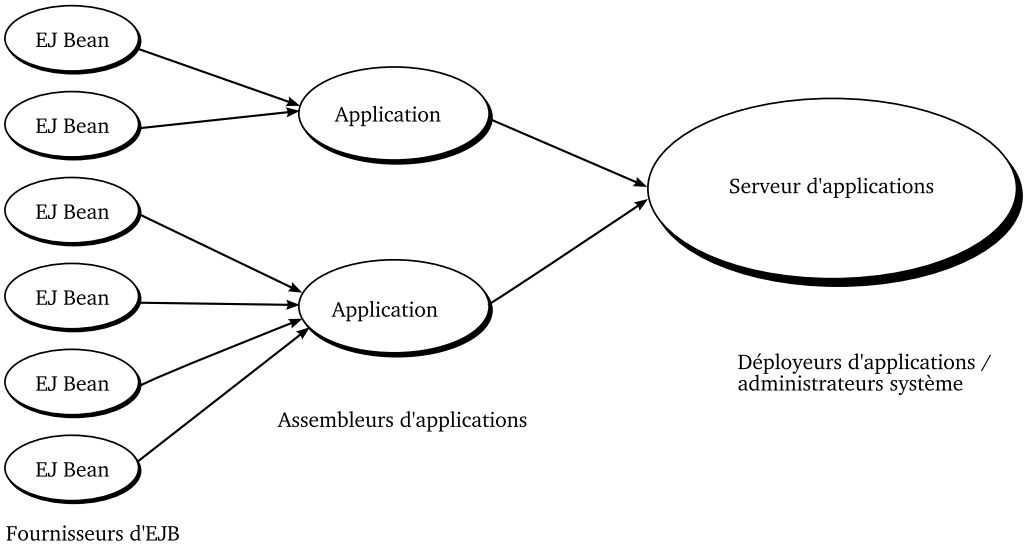


FIG. 3.9 – Relation entre les différents rôles

3.4.1 Modèle de sécurité

Nous venons de voir que la création d'une application à l'aide des EJBs suppose la participation de plusieurs acteurs avec des préoccupations différentes. Ainsi, pour que le modèle de sécurité fonctionne, un acteur dans un rôle ne doit pas prendre de décisions trop spécifiques qui restreindraient l'ensemble d'opérations d'un autre acteur. Par exemple, le fournisseur de Bean ne doit pas faire de supposition sur l'existence d'un mécanisme précis d'authentification étant donné que ce rôle incombe au déployeur d'applications qui devra choisir un mécanisme adapté à l'application.

Afin de supporter ce paradigme de programmation, l'architecture de sécurité des EJBs inclut les concepts de sécurité suivants :

- *Une sécurité basée sur les rôles* [22, 76]. Les fournisseurs de Beans et les assembleurs d'application assignent des privilèges de sécurité à des rôles et non pas à des utilisateurs. À leur niveau, ils ne peuvent que concevoir les divers mécanismes d'accès autour de divers rôles. L'assignation des rôles aux utilisateurs ne sera faite que par le déployeur d'application.
- *Sécurité déclarative*. Un assembleur d'application spécifie de manière déclarative, dans le descripteur de déploiement, qui (quels rôles) peut accéder à quels EJBs ou à quelles méthodes d'un EJB. Cette séparation de la logique de l'application et de la politique de sécurité permet de pouvoir modifier la politique de sécurité d'un EJB sans avoir à modifier son code source.
- *Sécurité par programmation*. Le développeur d'un EJB peut en utilisant une API accéder aux informations de sécurité de l'EJB. Il peut vérifier si l'utilisateur courant appartient à un groupe précis. Toutefois il se peut que le nom d'un rôle choisi par le concepteur de l'EJB ne corresponde pas au nom choisi par l'assembleur d'application. Cette situation résulte notamment de la séparation des préoccupations propre à l'infrastructure des EJBs. Cependant, le lien entre les deux rôles peut être spécifié dans un descripteur de déploiement.
- *Domaines de Protection*. Des entités peuvent vouloir communiquer sans passer par les mécanismes d'authentification. Un domaine de protection est un ensemble d'EJBs qui se font confiance mutuellement (figure 3.10). Quand un EJB interagit avec un autre au sein du même domaine de protection, aucune contrainte de sécurité n'est positionnée concernant l'identité de l'appelant. L'appelant peut propager son identité ou choisir une identité en fonction des contraintes d'identification de l'appelé. Les domaines de protection sont généralement liés au conteneur d'EJB. Pour les appels entrants, il est de la responsabilité du conteneur d'établir l'identité de l'appelant sous la forme d'un certificat X.509 ou d'un ticket Kerberos. En ce qui concerne les appels sortants, le conteneur est responsable de l'établissement de l'identité de l'appelant.

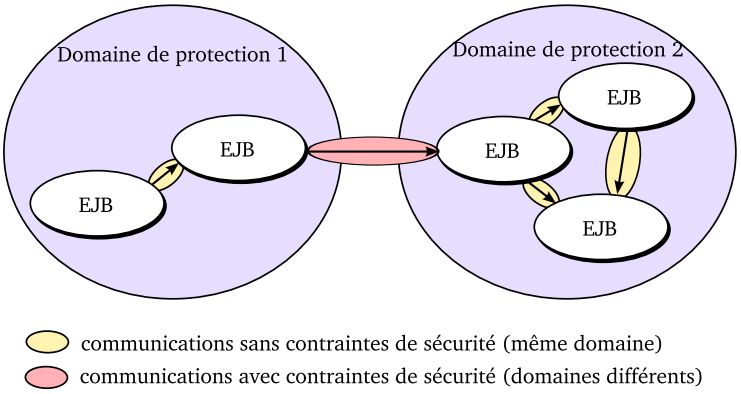


FIG. 3.10 – Domaines de protection

- *Propagation d'identité ou délégation*. L'invocation d'une méthode d'un EJB par un client authentifié, peut être réalisée sous l'identité de l'appelant (propagation) ou sous une identité indépendante de l'identité de l'appelant et configurée dans les descripteurs de déploiement (délégation).

3.4.2 Exemple

L'EJB que nous présentons maintenant gère sa sécurité de manière déclarative, il délègue à son conteneur toutes les opérations spécifiques à l'établissement des connexions sécurisées et à la validation des divers utilisateurs du système.

Nous présentons, dans un premier temps, le code de l'EJB (`EJBJournalIntime.java`) appelé à distance. Ce code est très simple étant donné que la seule action de sécurité est de vérifier que l'utilisateur appartient au bon rôle.

```
public String ajoutEntree(String message) {

  // test du role attribue a l'appelant
  // ctx represente le contexte de securite (gere par le conteneur
  // mais accessible par l'EJB)
  if (! ctx.isCallerInRole("redacteur"))
     throw new java.security.AccessControlException("L'utilisateur n'est pas dans
         le bon role.");

  // ecriture effective du message dans le journal
  this.ecrireEntree(message);
}
```

Nous supposerons que la définition des rôles et leurs associations avec les utilisateurs physiques sont effectives sur le serveur d'application. Cette supposition implique que le morceau de code XML suivant se trouve au sein du fichier de configuration du serveur d'application.

```
<security-role-mapping>
   <role-name>redacteur</role-name>
   <principal-name>arnaud</principal-name>
</security-role-mapping>
```

Le descripteur de déploiement de l'EJB comporte plusieurs parties relatives à la sécurité de notre EJB. Tout d'abord le descripteur doit exposer les rôles qu'il utilise.

```
<assembly-descriptor>
  <security-role>
      <role-name>redacteur</role-name>
  </security-role>
</assembly-descriptor>
```

Ensuite vient la définition des permissions accordées aux méthodes exposées par l'EJB.

```
<method-permission>
   <role-name>redacteur</role-name>
   <method>
       <ejb-name>JournalIntime</ejb-name>
       <method-name>ajoutEntree</method-name>
   </method>
</method-permission>
```

Une fois correctement configuré l'EJB peut être déployé sur le serveur d'application. Il apparaît que l'écriture d'un EJB sécurisé en utilisant la sécurité de façon déclarative reste aisée.

La partie cliente se repose sur le Java Authentification and Autorisation Service (JAAS) [87] en ce qui concerne le mécanisme d'authentification. Cette approche suppose l'écriture de plusieurs classes java et fichiers de configuration pour JAAS, toutes ces classes sont résumées par le tableau.

Fichier	**Description**
login.conf	fichier de configuration pour JAAS passé comme propriété lors du lancement du client
Client.java	Programme client utilisant JAAS pour se connecter
EJBCallbackHandler.java	Classe permettant la création d'un contexte de sécurité compatible avec la sécurité EJB
JournalIntimeAction.java	Classe comportant l'action à effectuer sur l'EJB JournalIntime

Nous présentons le code de la classe `Client.java` qui contient le code d'identification de l'utilisateur.

```
// creation du contexte de securite
LoginContext loginContext = new LoginContext("Sample", new
EJBCallBackHandler(utilisateur, motDePasse, url));

// identification de l'utilisateur
loginContext.login();

// recuperation du sujet (identite)
Subject sujet = loginContext.getSubject();

// creation de l'action
JournalIntimeAction action = new JournalIntimeAction(url,"mon message
de la journee");

// execution de l'action sous l'identite
Subject.doAs(sujet, action);
```

Les propriétés de chiffrement et confidentialité des messages échangés sont configurées au moment du lancement de l'application en passant les paramètres de sécurité nécessaires. Ces propriétés sont activées en encapsulant la couche de transport au sein d'une connexion SSL.

En conclusion, Les EJBs définissent une architecture à composants pour les applications distribuées basée sur le langage Java. La plupart des décisions concernant la sécurité sont exportées hors du code applicatif du composant et exprimées de manière déclarative au sein de descripteurs de déploiements. La confidentialité et l'intégrité des données entre le client et le composant EJB sont assurées par l'utilisation de SSL. Cependant, on peut regretter qu'il n'existe pas un mécanisme de sécurité similaire à celui fournit par un conteneur de sécurité pour la création de clients se connectant aux EJBs. Le client pourrait alors déléguer tous les mécanismes de sécurité à ce conteneur comme dans le cas des composants EJBs.

3.5 La plateforme .NET

La plateforme *.NET* et le langage de programmation *C#* sont la réponse de Microsoft à la plateforme J2EE de Sun et son langage de programmation Java. Contrairement aux débuts difficiles de Java , la plateforme .NET a dès le début été mise en avant par Microsoft et l'engouement de la part des développeurs fut rapide, notamment grâce aux environnements de développement créés ou adaptés pour .NET.

A l'instar de l'environnement Java, la plateforme .NET a pour but le développement d'applications portables et distribuées. L'utilisation d'une machine virtuelle, le *Common Language Runtime (CLR)* et d'un langage intermédiaire *MicroSoft Intermediaire Language (MSIL)* permettent d'envisager le portage de la plateforme sous d'autres systèmes d'exploitation, à l'instar du projet *Mono* qui développe une version GPL des standards .NET et s'exécutant sur Linux, Windows, OSX, BSD et Solaris. Le CLR possède son propre modèle d'exécution sécurisé. Du fait de la virtualisation de la plateforme d'exécution, le modèle de sécurité du CLR n'est pas limité par le système d'exploitation sous-jacent.

Les *assemblies* sont un élément fondamental de la programmation avec le Framework .NET et se composent généralement des quatre éléments suivants :

- le manifeste de l'assembly ;
- les métadonnées des types ;
- le code MSIL ;
- un ensemble de ressources nécessaires à l'application (fichiers images, données, ...).

L'*assembly* forme une unité de déploiement. Lorsqu'une application démarre, seuls les assemblies que l'application appelle initialement doivent être présents. Il forme aussi une limite de

sécurité. Un assembly correspond à l'unité au niveau de laquelle les autorisations sont demandées et accordées.

3.5.1 Modèle de sécurité

Le CLR et les classes de base de la plateforme jouent un rôle majeur dans les mécanismes de sécurité de la plateforme. Le CLR utilise plusieurs critères pour déterminer les permissions associées à un code allant être exécuté et obtient des informations de sécurité depuis des fichiers de configuration externes. Afin de couvrir le maximum de scénarios de sécurité, les mécanismes de sécurité introduisent plusieurs concepts.

Le code MSIL figurant dans un *assembly* ne sera pas exécuté si l'*assembly* ne possède pas de *manifeste*. Le *manifeste* contient les métadonnées qui permettent de résoudre les types et de satisfaire les demandes de ressources. Le manifeste spécifie également les types et les ressources exporté par l'assembly et énumère les assemblies dont il dépend. Afin de pouvoir exprimer des règles de sécurité sur ces unités, il faut pouvoir les identifier, on parle alors d'*evidence* (preuve) d'un assembly. Les preuves d'un *assembly* peuvent prendre les formes suivantes :

- le répertoire d'installation de l'*assembly* ;
- le `hash` (résumé) cryptographique de l'*assembly* ;
- la signature numérique de l'éditeur ;
- le `Strong Name` (identifiant unique d'une assembly) ;
- la zone de téléchargement (Internet, Intranet, machine locale) ;
- le site de téléchargement (nom de domaine) ;
- l'URL de téléchargement (adresse exacte).

Toutes ces preuves sont autant de critères qui permettent de paramétrer les permissions associées à un assembly.

Le mécanisme *Code Access Security* (CAS) permet d'identifier le niveau de confiance accordé à un morceau de code en fonction, par exemple, de sa localisation ou de son auteur. Il autorise le code de l'application à accéder aux ressources systèmes (système de fichiers, registres, réseau, bases de données, etc.). Elle met en œuvre un mécanisme de défense supplémentaire qui applique des limites à des parties de code. Il est ainsi possible de limiter les types de ressources auxquelles le code peut accéder. La figure 3.11 présente le fonctionnement du mécanisme de sécurité d'accès du code. Ce mécanisme est associé au concept d'*Evidence-based assemblies* dont le rôle est

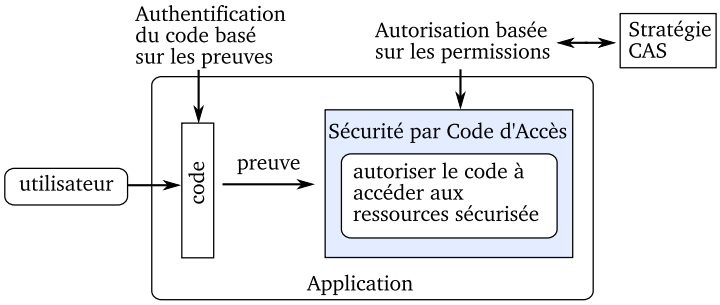


FIG. 3.11 – Code access security

d'examiner les assemblies chargées par le CLR avant leur exécution.

Du point de vue du mécanisme de sécurité, le rôle d'une *evidence* est d'être utilisée en combinaison avec les politiques de sécurité. Ces politiques servent au runtime à déterminer les permissions qui vont être associées à l'assembly. Ces politiques de sécurité sont configurables par les administrateurs systèmes et les utilisateurs. Il existe quatre niveaux d'affinement des permissions de chaque application.

Parmi les méthodes d'authentification présentes dans le CLR, nous pouvons citer celles du

système d'exploitation Microsoft Windows, l'authentification HTTP, Digest et Kerberos, ainsi que Microsoft Passport.

La sécurité peut être basée sur les concepts des *Rôles* au lieu d'utiliser directement les identifiants des utilisateurs. L'appartenance d'un utilisateur à un rôle est du ressort de l'administrateur de l'un des quatre niveaux d'affinements des politiques de sécurité.

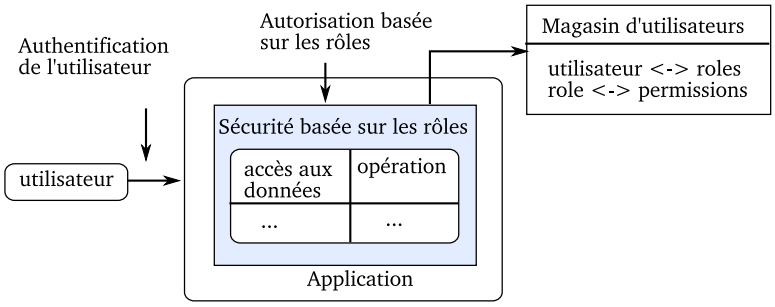


FIG. 3.12 – Sécurité utilisateur basée sur les rôles

Les permissions accordées à une action donnée correspondent à l'intersection des permissions des quatre niveaux de sécurité (tableau 3.1).

Type	Spécifié par	S'applique à
Entreprise	Administrateur	Tous les codes
Machine	Administrateur	Tous les codes sur la machine
Utilisateur	Administrateur et Utilisateur	Tous les codes associés à l'utilisateur
Application	Application	Tous les codes chargés par l'application

TAB. 3.1 – Les divers niveaux de politiques de sécurité en .Net

Afin de gérer le comportement de sécurité d'une assembly, le programmeur peut avoir recours à deux méthodes : la *sécurité déclarative* et la *sécurité impérative*.

La sécurité déclarative permet aux programmeurs de spécifier les critères de sécurité d'un assembly directement dans les métadonnées du son code. Les demandes d'autorisation et toutes les autres formes de sécurité déclarative sont spécifiées en tant qu'attributs personnalisés dans le code.

Les annotations des classes, des propriétés et des méthodes sont utilisées pour définir de manière plus précise les autorisations. La sécurité déclarative peut servir, par exemple, à vérifier que l'appelant d'une classe spécifique porte la signature d'un éditeur connu ou possède un nom fort particulier avant l'appel d'une méthode. Dans la mesure où les attributs déclaratifs font partis des métadonnées de l'assembly, ses besoins en matière de sécurité deviennent plus facilement identifiables. Des outils peuvent analyser les assemblies pour déterminer les méthodes avec demandes d'autorisations et celles avec assertions d'autorisations.

Les vérifications déclaratives conviennent particulièrement lorsque l'action et l'autorisation demandées sont déterminées au moment de la compilation. Si une méthode doit toujours vérifier un accès en écriture dans un répertoire par exemple, une vérification d'autorisation déclarative sera plus avantageuse. Si, par contre, l'emplacement de l'accès requis en écriture change, la sécurité impérative devient la solution la mieux adaptée. L'exemple ci-après présente le code C# responsable du contrôle d'accès, effectué de manière déclarative, à la méthode `Method`. Pour qu'un utilisateur puisse avoir accès à cette méthode, il doit appartenir au rôle `Responsable`.

```
[PrincipalPermission(SecurityAction.Demand, Role="Responsable")]
void Method()
{
// la methode est executee seulement si l'utilisateur
// appartient au role "Responsable"
```

```
}
```

La sécurité impérative est implémentée *directement dans le code*. Les programmeurs entreprennent des actions de sécurité par programme, et l'autorisation est accordée ou refusée selon l'état de la pile de sécurité. Lorsqu'une méthode émet une demande d'accès à un fichier spécifique, par exemple, cette demande peut échouer si l'appelant de la méthode (ou tout appelant de l'appelant) n'a pas reçu les autorisations nécessaires. La sécurité impérative étant implémentée par programme, les besoins en fonctionnalités dynamiques peuvent être satisfaits. La sécurité impérative est adaptée lorsque les autorisations ou les informations nécessaires ne sont connues qu'au moment de l'exécution de l'application.

```
void Method()
{
// on recupere le principal associe a la Thread courante
GenericPrincipal myPrincipal = Thread.CurrentPrincipal;
bool isInRole = myPrincipal.IsInRole("Responsable");
if ( isInRole )
 {
   // code effectue si l'appartenance de l'utilisateur
   // au role est confirmee
 }
}
```

3.5.2 Exemple

Pour la création de notre application "journal intime", nous utilisons *.NET Remoting* l'environnement pour applications distribuées de la plateforme .Net.

À ce titre, son rôle consiste à fournir toute l'infrastructure nécessaire à la distribution d'objets à l'aide de divers protocoles comme TCP ou SOAP. Si Remoting présente sur le papier une infrastructure solide, la construction d'une architecture n-tiers expose quelques lacunes qui semblent pourtant essentielles dans la création d'une application répartie. Nos remarques concernent tout d'abord la création des proxies vers les objets distants qui requiert le déploiement de l'assembly complète du serveur au sein de l'environnement du client. La deuxième remarque concerne l'absence d'un service de nommage qui impose la prise en charge de la localisation du serveur par le code de l'application cliente. Heureusement, il est possible de paramétrer la localisation au sein de fichiers de configuration externes.

.Net Remoting n'offre pas nativement de mécanismes de sécurité intégré, leur gestion est généralement déléguée à un serveur IIS *Internet Information Server* et à l'utilisation du protocole HTTPS. Cependant, dans ce cas précis, nous ne voulons pas nous reposer sur ce serveur afin d'avoir une application distribuée autonome pouvant gérer ses propres mécanismes de sécurité. La propagation d'un contexte de sécurité d'une application cliente doit donc être réalisée de façon *ad-hoc*. Cette approche est possible grâce à la structure d'une application utilisant .Net Remoting (figure 3.13). Entre l'objet appelant et l'objet appelé, il existe un ensemble d'objets intermédiaires permettant de masquer la distribution de l'objet serveur (*Remote Proxy*) mais aussi d'effectuer des traitements personnalisés sur les messages échangés (*MessageSink*). Les MessagesSink se composent d'une partie cliente et d'une partie serveur.

Quand un utilisateur déploie une application serveur, il peut référencer un fichier de configuration lui permettant tout d'abord de décrire son application mais aussi d'affiner son comportement notamment en précisant les divers MessageSink devant être utilisés.

Ingo Rammer, dans son ouvrage *Advanced .NET Remoting* [101] propose une gestion personnalisée de l'authentification basée sur l'introduction de ces intercepteurs spécialisés entre les objets communicants ainsi que sur l'utilisation d'un proxy modifié afin de capturer le contexte de sécurité de l'appelant. Si l'adaptabilité du mécanisme se révèle bonne via l'utilisation de fichiers externes, elle n'en demeure pas moins statique, le fichier de configuration d'un client devant être modifié à chaque fois que la configuration du serveur est modifiée.

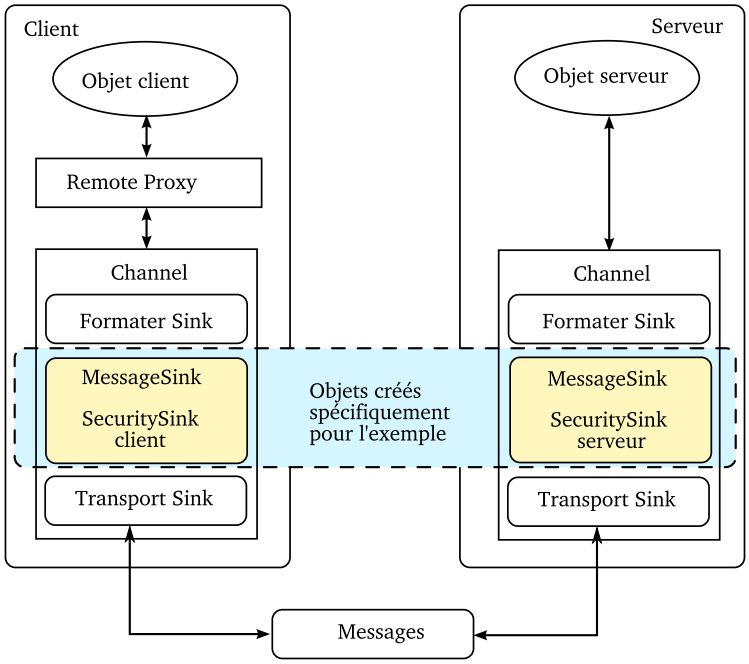


FIG. 3.13 – Architecture .Net Remoting avec gestion transparente de la sécurité

En conclusion, la modularité de la plateforme permet notamment d'utiliser des mécanismes d'authentification personnalisé qui authentifieraient automatiquement l'utilisateur de la partie cliente lors d'invocations sur le serveur. Cependant, même si cette solution apporte assez de souplesse pour intégrer notre vision d'une architecture sécurisée transparente, elle impose notamment la création de fichiers de configuration spécifiques ainsi que l'implantation de diverses classes représentant les divers *MessageSinks*. On peut regretter que des classes génériques fournissant les fonctions de base de la sécurité (authentification des utilisateurs, intégrité, confidentialité des messages) ne soient pas fournir de base avec le CLR.

3.6 Bilan

Dans ce chapitre, nous avons évalué les services de sécurité présents au sein d'un panel significatif d'intergiciels orientés applications distribuées dont Legion, Globus (OGSA), CORBA, la plateforme .NET et la spécification EJB. Le tableau 3.2 résume les caractéristiques de chaque intergiciel.

Cette étude a permis de mettre en valeur les approches et solutions proposées par chacun d'entre eux. Bien qu'ils n'adressent pas de la même façon les problèmes liés à la sécurité, ils permettent tous de mettre en place des applications sécurisées. Cependant, dans le contexte qui nous intéresse, à savoir le niveau de transparence des mécanismes de sécurité vis-à-vis du code source de l'application, tous ces intergiciels ne fournissent pas nativement les mécanismes nécessaires permettant sa mise en œuvre.

En effet, ces intergiciels se veulent les plus génériques possible afin de répondre aux attentes de la majorité des développeurs. Pour cela, ils intègrent le plus grand nombre de technologies relatives aux concepts de sécurité : cryptosystèmes à clés symétriques ou asymétriques, gestion des certificats (PKI), algorithme de hachages, ... Ces algorithmes et protocoles sont certes prouvés et vérifiés cependant ils n'ont pas été conçus pour supporter la dynamicité requise par les applications distribuées (création d'objets distants, changement de politiques de sécurité suite à la migration d'une activité, négociation dynamique de politique de sécurité, ...). Cette tâche est laissée au programmeur qui doit gérer par lui-même la distribution des clés, la définition de la politique de sécurité de chaque entité en présence, ...

De plus, l'intégration de ces protocoles au sein du code source des applications fige en quelque

	Legion	Globus	CORBA	EJB	.Net
Communications :					
Authentication	+	+	+	+	+
Intégrité	+	+	+	+	+
Confidentialité	+	+	+	+	+
Interception des communications	+	+	+	+	+
Niveaux de sécurité hiérarchiques	-	-	+	+	-
Politique de sécurité :					
Côté serveur	+	+	+	+	+
Côté client	-	-	+	-	-
Sécurité interne de l'application :					
configurable par le développeur	+	+	+	+	+
configurable par l'administrateur	+	+	+	+	+
configurable par l'utilisateur	-	-	-	-	-
Sécurité adaptable selon le déploiement	-	-	-	-	-

TAB. 3.2 – Tableau récapitulatif

sorte l'application dans un comportement déterminé qui ne pourra évoluer sans modifier le code source.

Nous en arrivons à la conclusion qu'il manque un modèle de sécurité qui permettrait d'adresser de manière transparente les problèmes de sécurité classiques inhérents aux applications distribuées, tout en laissant le développeur se concentrer sur le code de l'application. Parallèlement, ce modèle doit donner les outils nécessaires à l'utilisateur pour configurer comme il le souhaite la sécurité de l'application qu'il exécute.

Chapitre 4

Contexte : ProActive

ProActive est une bibliothèque Java pour le calcul parallèle, distribué et concurrent. La bibliothèque supporte la création d'objets distants, la mobilité des applications, les appels de méthodes asynchrones, des communications de groupe. Avec un ensemble réduit de primitives, *ProActive* fournit une API permettant la programmation d'applications distribuées pouvant être déployées aussi bien sur des réseaux locaux (LAN) que sur un ensemble de grilles de calcul interconnectées via Internet.

4.1 Modèle de programmation

La bibliothèque *ProActive* est conçue pour déployer des applications distribuées sur un nombre important d'ordinateurs à l'échelle planétaire si on le souhaite. On ne peut pas garantir l'uniformité ou la compatibilité des binaires entre tous les systèmes d'exploitation de ces ordinateurs. Le choix d'un language utilisant une machine virtuelle s'est dès lors imposé. Il permet de fournir un framework unifié sur lequel les développeurs vont pouvoir coder leurs applications sans avoir à se soucier de la compatibilité du code qu'ils produisent. C'est dans cette optique que le language Java a été retenu. De plus, pour respecter ce souci de portabilité, la bibliothèque *ProActive* ne repose que sur des outils java standards. *ProActive* ne modifie pas la sémantique du langage Java ni la machine virtuelle java (JVM, *Java Virtual Machine*). Il n'y a pas non plus de pré-traitement du code source de l'application ou de modification de ce dernier pendant l'exécution du programme. Les seules techniques employées sont l'utilisation d'un protocole à méta-objets utilisant des techniques de réflexion pour manipuler les différentes séquences d'évènements pouvant survenir lors de l'exécution d'une application en environnement réparti.

Il existait déjà des bibliothèques java permettant la création d'objets accessibles à distance telles que Java Remote Methode Invocation [113] ou encore Java IDL [73]. Cependant, les objets distants doivent soit être accédés au travers d'interfaces spécifiques, soit demander une profonde modification du code existant afin de transformer les objets locaux en objets distants. Il n'existe pas de polymorphisme entre les objets java standards et les objets accessibles à distance. C'est pourtant un prérequis essentiel qui permettrait au développeur de se concentrer sur l'amélioration de la structuration de son application plutôt que de s'occuper de tâches de bas niveau (distribution, équilibrage de charge, gestion de la migration, ...) qui devraient être gérées par l'intergiciel. Afin de donner une vue homogène de l'application, la bibliothèque *ProActive* se base sur le concept des objets actifs. La figure 4.1 montre comment une application initialement séquentielle et monothreadée peut évoluer en une application distribuée et multi-threadée sans changement dans la structure intrinsèque de l'application.

4.2 Objets Actifs

Une application construite avec la bibliothèque *ProActive* est composée d'entités de grain moyen appelées *objet actif* [19]. Etant donné un objet java standard, la bibliothèque permet de

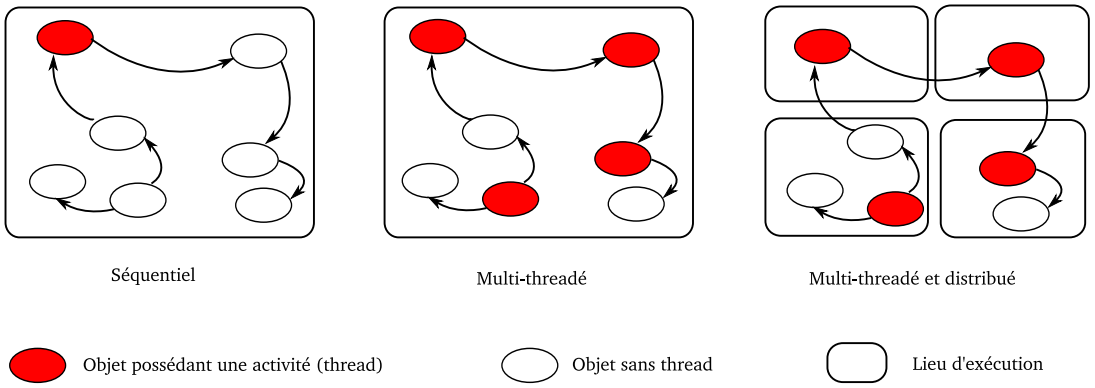


FIG. 4.1 – Du séquentiel au multi-threadé et distribué

lui rajouter, de manière transparente des comportements supplémentaires. On peut citer par exemple : la transparence vis-à-vis de la localisation physique de l'objet ou encore les mécanismes de synchronisation. Du point de vue applicatif, un objet actif est composé d'un objet racine possédant une activité et d'un graphe d'objets passifs (ne possédant pas d'activité). Le type de l'objet actif dépend de l'objet racine. l'ensemble contenant l'objet actif et le graphe d'objets passifs se nomme sous-système. Il est important de noter que le seul point d'entrée dans un objet actif est l'objet racine. La sémantique ne permet pas le partage entre des objets passifs : les seuls objets qui peuvent avoir une référence sur un objet passif d'un objet actif sont les objets appartenant au même objet actif. Un objet passif d'un object actif ne peut avoir de référence sur un objet passif d'un autre objet actif. Par contre, un objet passif peut avoir une référence sur un objet actif. Les diverses références possibles sont présentées sur la figure 4.2.

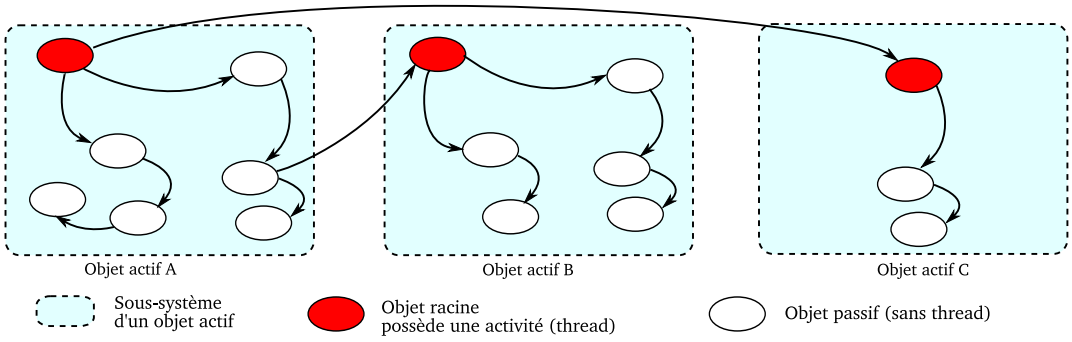


FIG. 4.2 – Compositions d'Objets actifs

4.2.1 Anatomie d'un objet actif

L'objet actif est une composition de plusieurs objets évoluant sur deux niveaux différents (figure 4.3). Le premier niveau nommé *de base* contient l'objet java standard rendu actif et son graphe d'objets passifs. Le second niveau appelé *méta-niveau* contient tous les objets qui vont rajouter des propriétés non fonctionnelles à l'objet. Ils ont été rajoutés à l'objet quand il a été rendu actif. La partie méta de l'objet actif est en charge d'assurer la transparence des communications, de gérer la file d'attente des appels de méthodes qui vont être exécutés sur l'objet rendu actif.
Les objets qui composent la partie méta d'un objet actif sont agencés de manière à séparer le plus distinctement possible les différents mécanismes définissant les divers comportements fonctionnels (migration, communication de groupe) ou non-fonctionnels (sécurité, tolérance aux pannes) d'un objet actif. Ainsi, à chaque service fourni par un objet actif correspond au moins un méta-

objet.

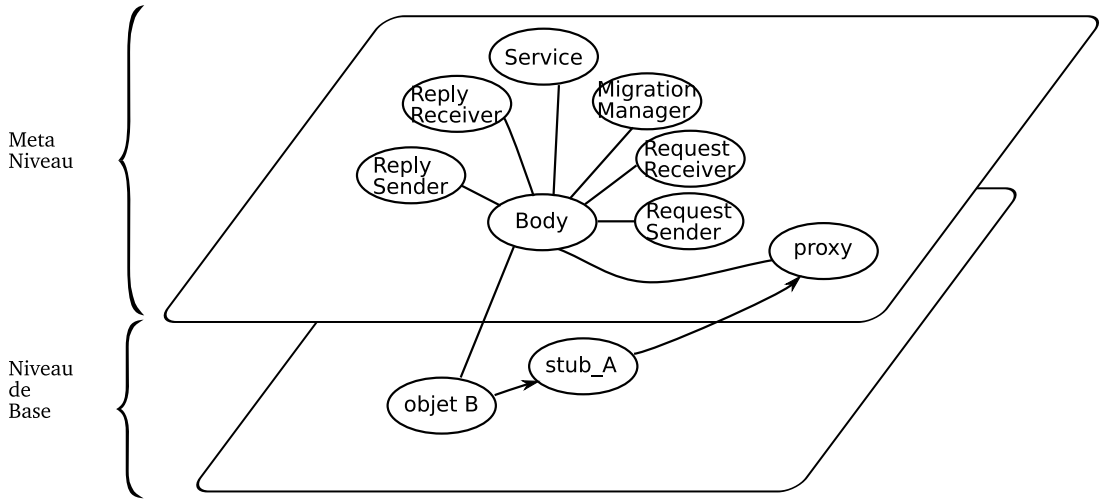


FIG. 4.3 – Anatomie d'un objet actif

À tout objet actif sont associés un *Proxy* et un *Body*. Le Body est le point d'entrée d'un objet actif. C'est la seule partie visible et accessible d'un objet actif. Le body sert de chef d'orchestre et gère le comportement de tous les autres méta-objets. L'ensemble Stub-Proxy est ce qu'on peut appeler une référence vers un objet actif. Il permet de masquer la notion de référence locale ou distante. Il sert à relayer les communications provenant d'un objet (actif ou non) vers un objet actif. Le stub est un objet java qui est polymorphiquement compatible avec l'objet rendu actif. Ce polymorphisme est assuré par héritage. Le stub est créé dynamiquement lors de la création d'un objet actif. Il est la représentation locale d'un objet distant.

La figure 4.4 présente les divers objets et méta-objets traversés lors d'un appel de méthode d'un objet actif sur un autre. Premièrement, le stub intercepte les appels de méthode vers l'objet actif qu'il représente et réifie l'appel de méthode (1). Réifier un appel de méthode consiste à transformer un élément abstrait (l'appel de méthode) en un objet concret afin de pouvoir le manipuler. Une fois l'appel de méthode réifié, il est envoyé au proxy (2) qui contient tout le code nécessaire pour transférer l'appel de méthode au body de l'objet actif cible. Notamment, le Proxy crée un objet de type *Requête* qui va encapsuler l'appel de méthode ainsi que d'autres paramètres qui permettent, entre autre chose, de gérer la transmission du résultat de l'appel de méthode s'il y en a un ou encore l'asynchronisme de cet appel. Le Proxy passe la Requête au body de l'objet appelant (2) qui le transmettra à son RequestSender (3). Le RequestSender est le méta-objet en charge de l'expédition de la requête au body. La requête est envoyée au body de l'objet actif distant (4). Quand un body reçoit une requête, il la passe à son RequestReceiver (5). Le RequestReceiver place ensuite la requête (6) dans la queue des requêtes. Le méta-objet Service représente l'activité de l'objet actif. Il contient la thread Java et sert les requêtes en les récupérant dans la queue des requêtes suivant le comportement qui lui a été défini. Si l'exécution d'une méthode renvoie une valeur de retour, il crée la réponse et la passe au méta-objet chargé de l'expédition des valeurs de retour. Symétriquement au RequestSender et au RequestReceiver, il existe des méta-objets qui gèrent l'envoi et la réception des réponses : le ReplySender et le ReplyReceiver.

4.2.2 Création d'un objet actif

L'instanciation d'un objet java standard se fait par l'instruction

```
A a = new A (2, "salut" );
```

Si nous voulons lui ajouter les fonctionnalités que proposent les objets actifs, il suffit seulement de modifier le code responsable de l'instanciation de cet objet. Il existe trois manières d'instancier un objet actif à la place d'un objet java standard.

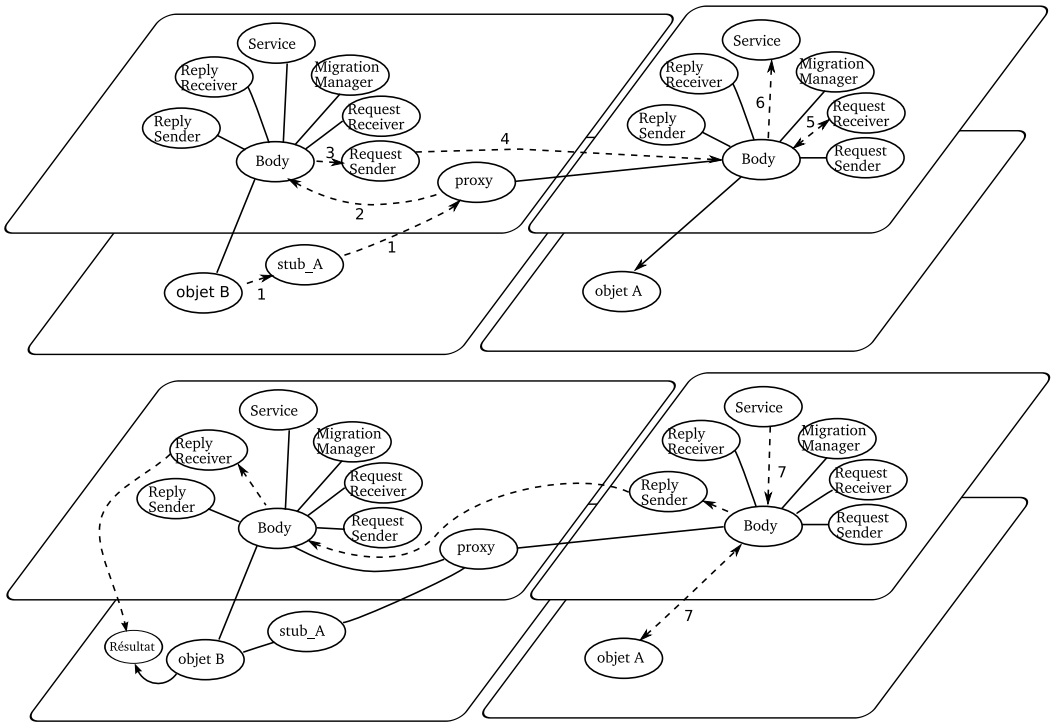


FIG. 4.4 – Communication entre deux objets actifs

1. L'approche *Class-based* est la plus statique. Elle implique la création d'une nouvelle classe qui implémente l'interface marqueur `Active` et hérite le cas échéant d'une classe préalablement existante. `Active` étant une interface marqueur pure, elle n'impose aucune contrainte au développeur. Il existe cependant des interfaces (`EndActive`, `InitActive`, `RunActive` étendant l'interface `Active`. Ces interfaces comportent des méthodes permettant de modifier le comportement par défaut des méta-objets de l'objet actif.
   Le code d'instanciation d'un objet actif selon l'approche *Class-based* est le suivant :

   ```
   public class PA extends A implements Active {...}

   Object[] params = { new Integer(2), "salut"};
   A a = (A) ProActive.newActive("PA", params, node);
   ```

   Le premier paramètre est le nom qualifié de la classe (nom du package + nom de la classe). Le deuxième paramètre est un tableau d'objets qui contient les paramètres à passer au constructeur lors de la création de l'objet actif. Le troisième paramètre spécifie le nœud sur lequel l'objet actif va être créé. Ce nœud peut être local ou distant.
2. L'approche *Instanciation-based* permet d'utiliser une classe java standard et d'instancier un objet actif sans avoir à modifier ni le code, ni le bytecode de la classe.

   ```
   Object[] params = { new Integer(2), "salut"};
   A a = (A) ProActive.newActive("A", params, node);
   ```

3. L'approche *Object-based* permet de rendre actif un objet java déjà instancié. Il est ainsi possible de rendre actif un objet java dont le source n'est pas disponible. Cette approche réalise une copie profonde de l'objet passé en paramètre qui continuera d'exister au sein de la machine virtuelle. Si des objets avaient des références sur l'objet avant son activation, les références pointeront toujours sur l'objet initial et non sur la copie rendue active.

   ```
   A a = new A (2, "salut" );
   a = (A) ProActive.turnActive(a, node);
   ```

4.3 Sémantique des communications

L'instanciation d'un objet actif retourne un objet polymorphiquement compatible avec le type de la classe d'origine. L'objet retourné est en fait une sous-classe de la classe d'origine. Cela nous permet de rendre transparente la distribution des objets actifs. Un objet java peut ne pas avoir conscience qu'il s'adresse à un objet distant, il fera des appels sur les méthodes publiques de l'objet actif comme si ce dernier était un objet java standard et local à la machine virtuelle.

La communication entre objets actifs est assurée par un échange de messages. Les messages sont des objets java sérialisables. On peut les comparer aux paquets utilisés par le protocole TCP. En effet, ces messages contiennent (1) une partie servant à leur routage par les divers éléments de la bibliothèque, (2) une partie contenant les données devant être communiquées à l'objet distant. Si les possibilités de faire des appels de méthodes restent les mêmes, la sémantique de communication associée à une méthode variera selon la signature de cette dernière.
Le tableau 4.1 présente les différentes sémantiques de communication selon les différentes signatures des méthodes.

Sémantique	Signature de la méthode	Nombre de messages échangés
One-way	La méthode n'a pas de valeur de retour (void) et ne lève pas d'exception.	Un message envoyé de l'appelant vers l'appelé.
Synchrone	– La méthode retourne un objet non reifiable : type primitif ou classe final, – La méthode peut lever une exception : on doit bloquer l'appelant pour éviter qu'il ne sorte du bloc `try/catch`	Deux messages échangés, un lors de l'appel de méthode, le deuxième lorsque la méthode a été exécutée afin de renvoyer le résultat de l'appel de méthode. La thread de l'appelant est bloquée jusqu'à réception du résultat de l'appel de méthode.
Asynchrone	Ne correspond ni au cas synchrone, ni au cas one-way : types réifiables, aucune exception	Deux messages échangés, un lors de l'appel de méthode, le deuxième lorsque la méthode a été exécutée afin de renvoyer le résultat de l'appel de méthode.

TAB. 4.1 – Signature des méthodes et sémantique des appels

Dans tout échange de messages, il existe une phase de *rendez-vous* lors de l'envoi du message. Ce rendez-vous assure à l'appelant que le message a été correctement délivré à l'objet actif appelé et qu'il a été placé dans la file d'attente des requêtes que l'objet actif appelé doit servir.

4.4 Migration des activités

La mobilité d'une application est un paradigme de programmation qui consiste à donner la possibilité aux divers éléments d'une application de changer de localisation en cours d'exécution [60]. La migration permet la migration des objets actifs d'un noeud vers un autre nœud. La migration telle qu'elle a été implantée au sein de *ProActive* est dite *faible* ; le code est mobile mais pas le contexte d'exécution. Le concept de migration faible s'oppose au concept de migration forte. La migration dite forte permet la migration d'un objet à n'importe quel moment pendant l'exécution de l'objet. Cette restriction est due au langage Java qui ne permet pas la capture des contextes d'exécution des Threads. La conséquence immédiate est l'impossibilité de faire migrer un objet tant que ce dernier n'a pas atteint un état désigné comme stable.

N'importe quel objet actif a la possibilité de migrer ou d'être migré. Si l'objet actif possède un graphe d'objets passifs, ces derniers seront déplacés également vers la nouvelle localisation. Le processus de migration repose actuellement sur la sérialisation Java pour transférer les objets

d'une localisation vers une autre. C'est pour cela que tous les objets (actifs et passifs) doivent implanter l'interface marqueur `Serializable` afin que le processus de sérialisation puisse les transférer.

Le processus de migration peut être déclenché soit par l'objet actif lui-même, soit par un objet extérieur à l'objet actif. Il existe une API permettant de programmer la migration d'une activité. Cette API est constituée d'une seule méthode nommée `migrateTo` ; méthode statique de la classe `ProActive`. Afin de faciliter l'utilisation de la migration, la méthode `migrateTo` est surchargée pour former deux ensembles de méthodes statiques.

Le premier ensemble réunit les méthodes qui seront appelées à l'intérieur de l'objet actif. Ces méthodes reposent sur l'assertion que la thread appelante est celle de l'objet actif. La signature de ces méthodes est la suivante :

```
// Migration vers le noeud designe par l'URL nodeURL
migrateTo(String nodeURL)

// Migration vers le noeud node
migrateTo(Node node)

// Migration vers le noeud qui heberge l'objet actif activeObject
migrateTo(Object activeObject)
```

Le second ensemble de méthodes réunit les méthodes qui seront appelées de l'extérieur de l'objet actif. Dans ce cas, l'objet appelant doit posséder une référence sur le `Body` de l'objet actif qu'il veut faire migrer.

```
// Migration de bodyToMigrate vers le noeud node
migrateTo(Body bodyToMigrate, Node node, boolean priority)

// Migration de bodyToMigrate vers le noeud node
migrateTo(Body bodyToMigrate, String nodeURL, boolean priority)

// Migration de bodyToMigrate vers le noeud qui heberge l'objet
//actif activeObject
migrateTo(Body bodyToMigrate, Object activeObject, boolean priority)
```

Le paramètre `priority` permet de modifier le comportement de prise en compte par l'objet actif du processus de migration. Si le paramètre est positionné à vrai, la requête de migration sera prioritaire sur toutes les requêtes déjà mises en attente dans la queue des requêtes de l'objet actif. Si le paramètre est positionné à faux, la requête sera traitée en priorité normale, c'est-à-dire qu'elle sera ajoutée à la suite des requêtes déjà en attente au sein de l'objet actif.

La migration d'une activité pose le problème de connectivité entre les objets. Le processus de migration doit assurer que la migration d'un objet actif n'entraînera pas des pertes de connectivité. Si un objet possède une référence sur un objet actif, la migration de ce dernier doit être faite de manière à ce que l'objet qui n'a pas migré puisse continuer de communiquer avec l'objet qui a migré. Le deuxième problème est celui de la localisation d'un objet actif. On doit pouvoir retrouver un objet qui a migré et cela même si, initialement, on ne possédait pas de référence sur celui-ci. Pour palier ces problèmes, deux solutions ont été introduites au sein de la bibliothèque : (1) les *répéteurs* et le *serveur de localisation*. Un répéteur est une référence laissée par l'objet actif au moment de la migration. Ce répéteur pointe vers le nouvel emplacement de l'objet actif. Il reçoit les messages à la place de l'objet qui a migré et les renvoie à ce dernier. Ce mécanisme est mis en évidence par la figure 4.5. Si l'objet migre plusieurs fois, on assiste à la création d'une chaîne de répéteurs. Plus la chaîne est longue, plus la probabilité qu'un des maillons de la chaîne disparaisse suite à des problèmes réseaux ou des problèmes sur la machine contenant le maillon augmente. Les longues chaînes entraînent une baisse des performances à cause du nombres de sauts à effectuer. La formation de chaînes de répéteurs est donc à éviter. La méthode utilisée par *ProActive* s'appelle le *tensioning* et consiste lors du premier appel de méthode après migration à remplacer le lien vers le premier élément de la chaîne des répéteurs par un lien direct vers le

nouvel emplacement de l'objet actif.

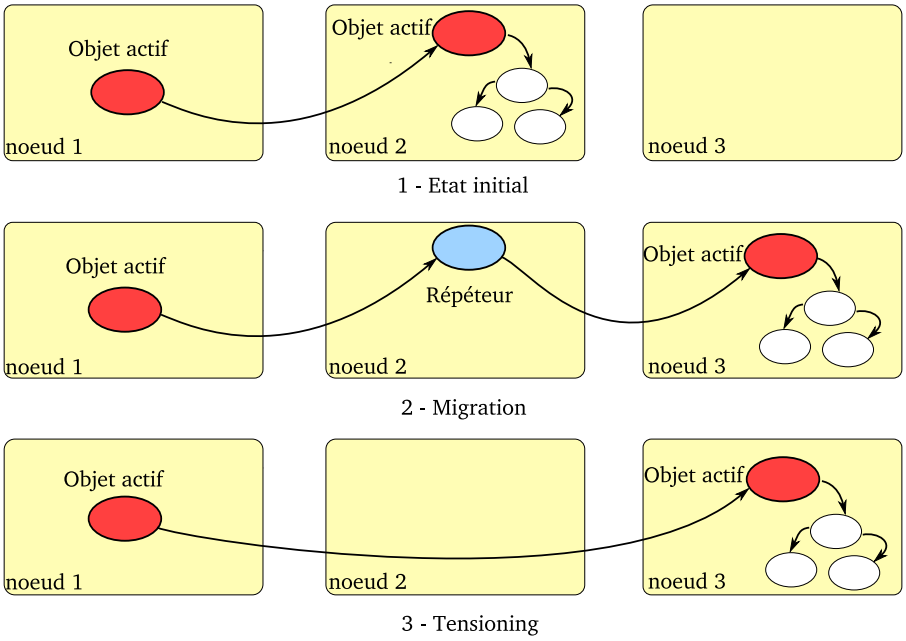


FIG. 4.5 – Migration avec répéteurs

Le second mécanisme se base sur un serveur de localisation (figure 4.6) qui possède une référence sur chaque objet actif présent dans le système. A chaque migration, l'objet actif envoie sa nouvelle localisation au serveur. Après la migration d'un objet actif, les références que pouvaient posséder d'autres objets sur celui-ci deviennent invalides. Lors du premier appel de méthode sur un objet qui a migré l'appel va échouer. A ce moment, le mécanisme de localisation va de manière transparente : (1) contacter le serveur de localisation afin de connaître le nouvel emplacement de l'objet actif qui a migré, (2) mettre à jour la référence qu'il possédait, (3) effectuer de nouveau l'appel de méthode. Contrairement à l'approche par répéteurs, on introduit des messages supplémentaires envoyés par l'objet actif qui a migré et par le mécanisme de localisation lors de l'échec d'une communication.

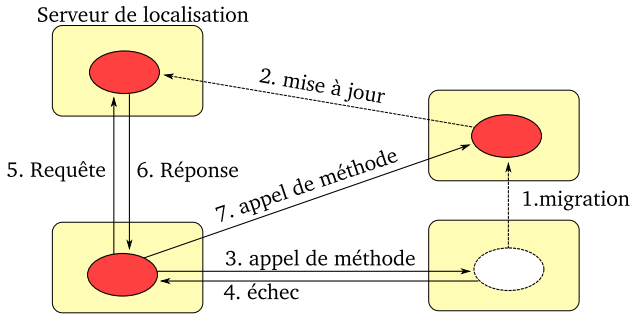


FIG. 4.6 – Migration avec serveur de localisation

4.5 Communications de groupe typé

Le système de communications de groupe existant dans la bibliothèque repose sur le mécanisme élémentaire d'invocation distante et asynchrone de méthodes. Comme l'ensemble de la

bibliothèque, ce mécanisme est mis en application en utilisant une version standard de Java. Le mécanisme de groupe est indépendant de la plateforme. Il doit être considéré comme une réplique de plusieurs invocations à distance de méthode vers des objets actifs. Naturellement, le but est d'incorporer quelques optimisations à l'exécution, de façon à réaliser de meilleures exécutions qu'un accomplissement séquentiel de n appels de méthode à distance. De cette façon, ce mécanisme est la généralisation du mécanisme d'appel de méthode asynchrone sur des objets distants.

La disponibilité d'un tel mécanisme de communication de groupes simplifie la programmation des applications en regroupant les activités semblables fonctionnant en parallèle. En effet, du point de vue de la programmation, utiliser un groupe d'objets du même type, appelé *groupe typé*, prend exactement la même forme que l'utilisation d'un simple objet de ce type. Ceci est possible grâce aux techniques de réification introduites précédemment.

Les groupes sont créés en utilisant la méthode statique :

```
ProActiveGroup.newGroup("ClassName", paramètres[], nœuds[]);
```

La superclasse commune à tous les membres du groupe doit être indiquée à la création du groupe, et lui donne ainsi un type minimal. Les groupes peuvent être créés vides, puis remplis par des objets actifs déjà existants. Des groupes non-vides peuvent aussi être construits en utilisant deux paramètres supplémentaires : une liste de paramètres requis pour la construction des membres du groupes et la liste des nœuds où ils seront créés. Le n-ième objet actif est créé avec les n-ièmes paramètres sur le n-ième nœud. Dans ce cas, le groupe est créé et les objects actifs sont construits puis immédiatement inclus dans le groupe. Prenons le cas d'une classe standard Java :

```
class A {
 public A() {}
 public void foo () {...}
 public V bar (B b) {...}
}
```

Voici un exemple de la création de deux groupes :

```
 // Pre-construction de parametres pour la creation d'un groupe
 Object[][] params = { {...} , {...} , ... };
 // Noeuds sur lesquels seront crees les objets actifs
 Node[] nodes = { ... , ... , ... };
 // Creation 1:
 // Creation d'un groupe vide de type "A"
 A ag = (A) ProActiveGroup.newGroup("A");
 // Creation 2:
 // Un groupe de type "A" et ses membres sont crees en meme temps
 A ag2 = (A) ProActiveGroup.newGroup("A", params, nodes);
}
```

Des éléments ne peuvent être inclus dans un groupe que si leur type est compatible avec la classe spécifiée à la création du groupe. Par exemple, un objet de classe B (B étendant A) peut être inclus dans le groupe. Cependant, étant basées sur le type de A, seules les méthodes définies dans la classe A peuvent être appelées sur le groupe, mais notons que la redéfinition de méthode va fonctionner normalement.

La représentation typée des groupes correspond à la vue fonctionnelle des groupes d'objets. Afin de fournir une gestion dynamique des groupes, une deuxième (et complémentaire) représentation d'un groupe a été conçue. Cette seconde représentation suit un modèle plus standard : l'interface `Group` étend l'interface `Collection` de Java qui fournit des méthodes telles que `add`, `remove`, `size`, ... Cette gestion de groupes comporte une sémantique simple et classique (ajouter dans le groupe, enlever le n-ième élément, ...) qui fournit une propriété de rang des éléments au sein d'un groupe. Les méthodes de gestion d'un groupe ne sont pas disponibles dans la *représentation typée* mais dans la *représentation de groupe*. La double représentation est un choix de conception. Nous avons choisi l'association de deux représentations complémentaires, l'une pour l'usage fonctionnel et l'autre pour la gestion dynamique. Au niveau de l'implémentation, il a été pris soin de maintenir une cohérence forte entre les deux représentations d'un même groupe. Les

modifications faites sous une forme sont instantanément reportées sous l'autre forme. Pour alterner d'une forme à l'autre deux méthodes existent (voir figure 4.7) : la méthode statique `getGroup` de la classe `ProActiveGroup` retourne la représentation de groupe à partir d'une représentation typée. La méthode `getGroupByType` définie dans l'interface `Group` effectue l'opération inverse.

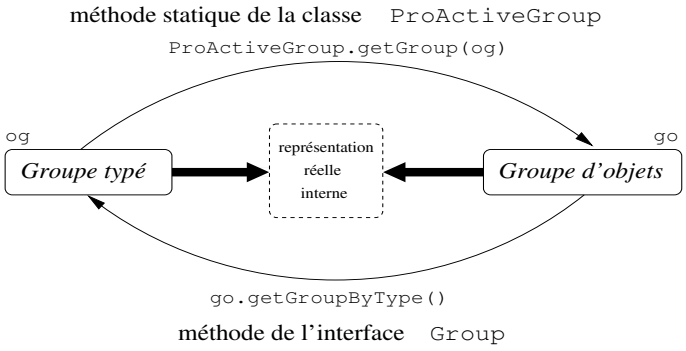


FIG. 4.7 – Double représentation des groupes

Voici un exemple de l'emploi de chaque représentation d'un groupe :

```
  // Creation d'un objet Java standard et de deux objets actifs
A a1 = new A();
A a2 = (A) ProActive.newActive("A", paramsA[], node);
B b = (B) ProActive.newActive("B", paramsB[], node);
  // Notons que B etend A
  // Pour la gestion d'un groupe, on obtient la representation
  // de groupe a partir d'un groupe type
Group gA = ProActiveGroup.getGroup(ag);
  // On ajoute les objets au groupe
gA.add(a1);
gA.add(a2);
gA.add(b);
  // Une nouvelle reference vers le groupe type peut etre obtenue
A ag_new = (A) gA.getGroupByType();
}
```

Notons que les groupes ne contiennent pas nécessairement que des objets actifs, mais peuvent contenir des objets standard de Java et des groupes typés (d'où l'obtention de groupes hiérarchiques). La seule restriction est qu'ils soient de classe compatible avec la classe du groupe. La section suivante examinera les implications de tels groupes hétérogènes dans la gestion des communications vers les éléments du groupe.

L'invocation d'une méthode sur un groupe a une syntaxe identique à une invocation de méthode sur un objet Java :

```
// Une communication de groupe
ag.foo();
```

Bien sûr, un appel de ce type a une sémantique différente : l'appel de méthode est rendu asynchrone et est propagé vers tous les membres du groupe. Un appel de méthode sur un groupe est un appel de méthode sur chaque membre du groupe. Ainsi, si un membre est un objet actif, la sémantique de communication de *ProActive* sera utilisée, s'il s'agit d'un objet Java, la sémantique sera celle d'un appel de méthode classique.
Par défaut, les paramètres de la méthode invoquée sont diffusés à tous les membres du groupe (*broadcast*). Il est également possible, grâce à des méthodes statiques, de changer le comportement des groupes pour que les paramètres soient distribués selon les membres (*scatter*) et non plus diffusés : pour distribuer les données à travers une communication de groupe, il suffit de rassembler ces données au sein d'un groupe et de le passer en paramètre à un appel de méthode.

Voici un exemple :

```
// Creation du groupe de parametres
B bg = (B) ProActiveGroup.newGroup("B", {{...},{...},...} , {...,...,...});
// bg est envoye a tous les membres de ag
ag.bar(bg);
// Changement du mode de communication de ag (\emph{distribution})
ProActiveGroup.setScatterGroup(bg);
// Chaque membre de bg est envoye a un membre de ag
ag.bar(bg);
}
```

Pour profiter du modèle d'appel distant et asynchrone de *ProActive*, quelques nouveaux mécanismes de synchronisation ont été ajoutés. Des méthodes définies sur l'interface `Group` permettent d'exécuter diverses formes de synchronisation (`waitOne`, `waitN`, `waitAll`, `waitTheNth`, `waitAndGet`, ...).
Par exemple :

```
// Une methode appelee sur un groupe type
V vg = ag.bar(...);
// On accede a la representation Group de vg
Group gV = ProActiveGroup.getGroup(vg);
// Pour attendre et retourner le premier membre de vg
V v = (V) gV.waitAndGetOne();
// Pour attendre que tous les membres de vg soient arrives
gV.waitAll();
}
```

4.6 Déploiement des applications

Déployer une application consiste à créer et configurer toute l'infrastructure nécessaire à l'exécution de l'application. Cependant, les contraintes et les besoins d'une application peuvent varier selon le déploiement choisi. L'approche par *descripteurs de déploiement* présentée dans cette section permet de résoudre les problèmes de configuration de l'application lors de son déploiement. Le slogan de Sun concernant Java était « *Compile once, run everywhere* »[1], celui de *ProActive* pourrait être « *Compile once, deploy and run everywhere* »[2].

4.6.1 Limitations

Lors de l'instanciation d'un objet actif, il est possible de spécifier sa localisation future. Pour identifier les différents emplacements pouvant contenir des objets actifs, on utilise le concept de *Nœud*. Un nœud est un objet *ProActive* dont le but est de créer une entité logique capable de contenir des objets actifs [10]. Un nœud est identifié par une URL (une chaîne de caractères) de la forme suivante : `protocole://machine:numero_de_port/Nom_du_Noeud`. Par exemple, l'URL `rmi://noadcoco.inria.fr:2026/Noeud1` indique que le nœud Noeud1 est accessible en ouvrant une connexion en utilisant le protocole rmi vers la machine noadcoco.inria.fr sur le port 2026.

Un nœud est toujours contenu dans un *Runtime*. Un runtime est un objet Java accessible à distance qui offre un ensemble de services (création de nœuds, d'objets actifs, ...) permettant l'accès à des machines virtuelles distantes. Un runtime est aussi identifié par une URL qui possède la même syntaxe que celle des nœuds.

[1] Compiler une fois, exécuter partout
[2] Compiler une fois, déployer et exécuter partout

L'instruction suivante permet l'instanciation d'un objet actif étendant la classe java standard "A" et localisé sur le nœud `Node1` sur la machine noadcoco.inria.fr :

```
// Creation d un objet actif sur le noeud Node1 localise sur
// la machine noadcoco.inria.fr
A a = (A) ProActive.newActive("A", params,"rmi://noadcoco.inria.fr/Node1" );
```

Si le paramètre node est null ou s'il est omis, l'instanciation se fera sur la machine virtuelle courante :

```
// Creation d'un objet actif sur la machine virtuelle courante
// dans le noeud courant
A a = (A) ProActive.newActive("A", params, null);
A a = (A) ProActive.newActive("A", params);
```

Il est aussi possible de passer en paramètre un objet de type Node qui représente un nœud local ou distant :

```
// Creation d'un objet actif sur le noeud represente par
// l'objet Node node
A a = (A) ProActive.newActive("A", params, node);
```

Seule la dernière méthode possède une signature assez générique pour permettre de changer facilement le nœud qui accueillera le futur objet actif. Il nous faut maintenant un moyen de changer selon nos besoins la valeur de la variable node sans avoir à toucher au code source afin de le recompiler, ni au bytecode. Par exemple, on veut pouvoir utiliser des protocoles variés tels que rsh, ssh, LSF ou Globus pour créer les runtimes (JVMs) nécessaires à l'application. De la même manière, la découverte de ressources déjà existantes ou l'enregistrement de celles créées par l'application pourrait être fait via l'utilisation d'annuaires variées comme le RMIregistry, Jini, Globus, LDAP, etc.

4.6.2 Nœuds virtuels et descripteurs de déploiement

Nous voulons obtenir un code applicatif portable au niveau du déploiement et dans lequel nous avons retiré toutes les informations liées au déploiement physique des nœuds (protocoles de connexion, noms des machines, numéros de port, protocoles d'enregistrement et de recherche). Pour cela, nous allons distinguer les différents nœuds en leur attribuant un nom symbolique représenté par une chaîne de caractères. Le nom symbolique associé à un ou plusieurs nœuds porte le nom de *nœud virtuel*. De la même manière que les nœuds physiques qui permettent une structuration bien précise de l'application, le nœud virtuel permet de structurer l'application en fonction d'une architecture virtuelle. Cette approche offre aussi la possibilité de décrire l'application de manière abstraite, en terme d'activités conceptuelles.

Une fois l'application créée et architecturée autour des nœuds virtuels, on peut se pencher sur la phase de déploiement de l'application. Le déploiement d'une application se fait en lui passant en paramètre le *descripteur de déploiement* qui comme son nom l'indique va indiquer comment créer l'architecture dont l'application a besoin et où déployer les divers éléments constituant l'application. L'association entre les nœuds virtuels et les nœuds (le *mapping*) est spécifiée dans le descripteur de déploiement. Les opérations pouvant être configurées dans un descripteur de déploiement sont les suivantes :

- l'association des nœuds virtuels sur des nœuds devant être créés ou acquisition de nœuds déjà existants ;
- la manière de créer de nouveaux runtimes ou d'acquérir des runtimes existants qui accueilleront les futurs nœuds ;
- la manière de créer ou d'acquérir des nœuds virtuels.

Pour résumer, un nœud virtuel est :

1. identifié par un nom ;
2. utilisé dans le code source ;
3. défini et configuré dans un descripteur de déploiement.

un nœud virtuel, après déploiement, est associé à un ou plusieurs nœud réels. Le processus de déploiement est présenté dans la figure 4.8.

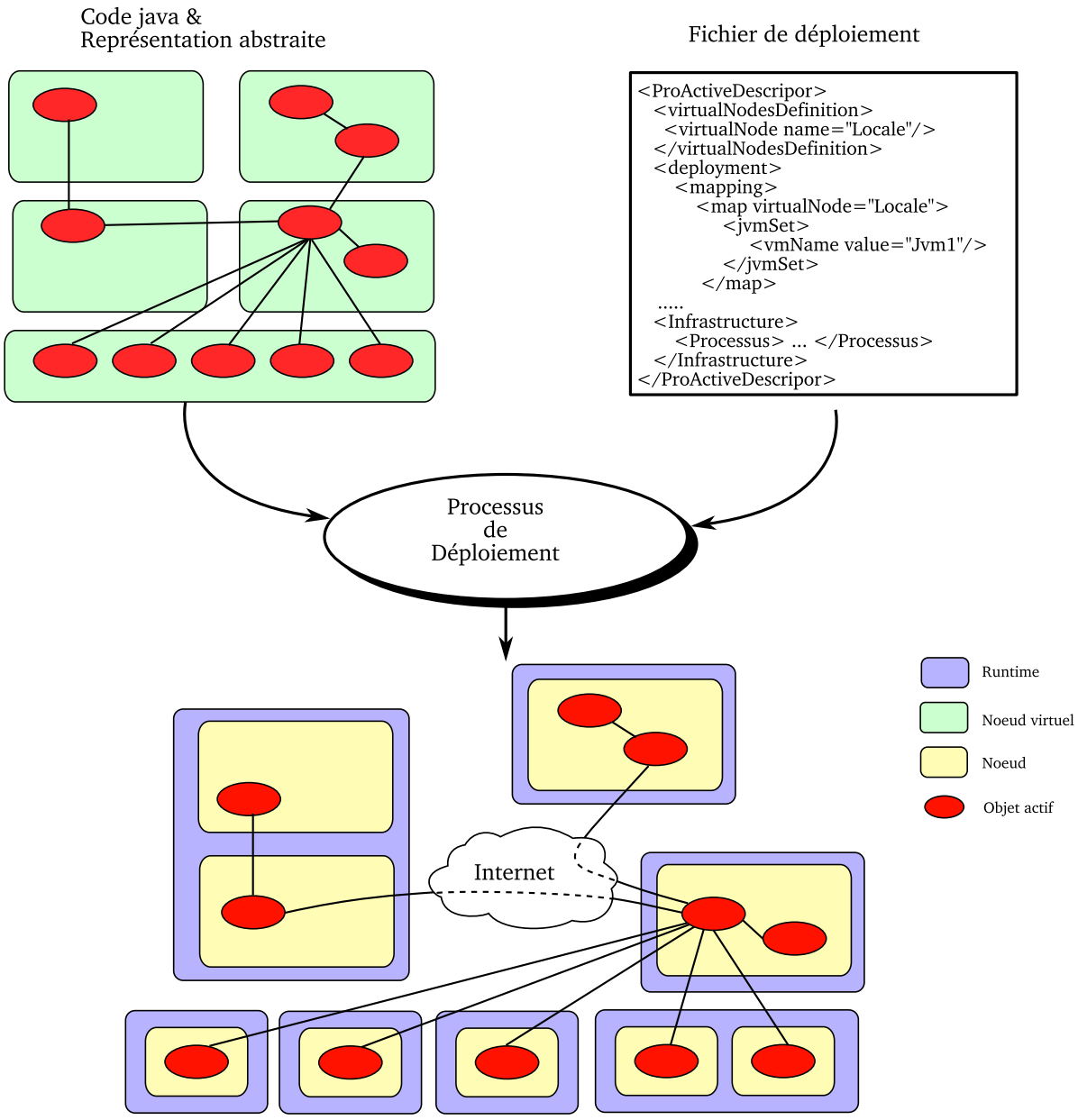


FIG. 4.8 – Processus de déploiement

Nous allons maintenant présenter et commenter le code d'une application utilisant les descripteurs de déploiement. Il s'agit de C3D un moteur de rendu 3D collaboratif et distribué. Les moteurs de rendu (Renderer) sont créés dans les nœud contenus dans le nœud virtuel VN_Renderer. Un objet central, le dispatcher créé sur le nœud virtuel VN_Dispatcher, synchronise et gère les moteurs permettant ainsi aux utilisateurs de se connecter à l'application, de visualiser et d'éditer collaborativement une scène 3D.

Dans la méthode `main` du programme, nous activons le descripteur de déploiement :

```
// descripteur.xml est le nom du fichier de deploiement

ProActiveDescriptor proActiveDescriptor = ProActive.getProactiveDescriptor("file:
    descripteur.xml");

// on active le descripteur.
proActiveDescriptor.activateMappings();
```

Lors de son activation, le descripteur de déploiement instancie tous les runtimes et nœuds nécessaires au fonctionnement de l'application. Il est alors possible de récupérer un objet représentant un nœud virtuel donné. Parmi les méthodes présentes dans la classe `VirtualNode`, la méthode `getNode()` permet de récuperer la référence sur un des nœuds contenu dans le nœud virtuel.

```
// Renderer et Dispatcher sont les noms des noeuds virtuels
// decrit dans le fichier de deploiement

VirtualNode renderer = proActiveDescriptor.getVirtualNode("VN_Renderer");
VirtualNode dispatcher = proActiveDescriptor.getVirtualNode("VN_Dispatcher");

// on recupere le premier noeud existant dans le
// noeud virtuel "dispatcher"
Node node = dispatcher.getNode();
```

Il est possible maintenant d'instancier un objet actif sur un nœud donné

```
// on cree un objet actif sur ce premier noeud
Dispatcher dispatcher = (Dispatcher) ProActive.newActive("Dispatcher",param, node)
    ;
```

Une autre possibilité offerte est la création d'un groupe d'objet juste en passant un nœud virtuel en paramètre. Le nombre d'objets créés est lié au nombre de nœuds contenus dans le nœud virtuel ce qui depend de la configuration du fichier de déploiement.

```
// on cree un moteur de rendu sur chaque noeud du noeud virtuel
// le resultat sera un groupe d'objet de type renderer
Renderer renderers = (Renderer) ProActive.newActiveAsGroup("Renderer",params,
    renderer);
```

Nous allons maintenant nous pencher sur l'étude du descripteur de déploiement associé au programme. Un descripteur de déploiement se divise en plusieurs grandes parties ayant chacune un rôle spécifique.

La première partie permet de définir les nœuds virtuels qui vont être utilisés par le programme ainsi que les propriétés de chacun des nœuds virtuels. La propriété `multiple` permet de spécifier que le nœud virtuel `Renderer` va pouvoir être associé à plusieurs runtimes (machines virtuelles java). Lorsque le nœud virtuel sera activé, il possédera un nœud sur chaque runtime qui lui a été associé.

```
<virtualNodesDefinition>
      <virtualNode name="Renderer" property="multiple"/>
      <virtualNode name="Dispatcher"/>
</virtualNodesDefinition>
```

La partie `deployment` permet d'associer des runtimes aux nœuds virtuels. On associe le nœud virtuel `Renderer` avec trois runtimes désignés par les balises `<vmName>`. Le nœud `Dispatcher` n'est associé qu'à un seul runtime.

```
<mapping>
      <map virtualNode="Renderer">
            <jvmSet>
                  <vmName value="Jvm1"/>
                  <vmName value="Jvm2"/>
                  <vmName value="Jvm3"/>
            </jvmSet>
```

```
      </map>
      <map virtualNode="Dispatcher">
            <jvmSet>
                  <vmName value="Jvm1"/>
            </jvmSet>
      </map>
</mapping>
```

Il faut ensuite lier les runtimes à des processus réels.
Les runtimes `Jvm1` et `Jvm2` seront des processus créés lors de l'activation du descripteur de déploiement.

```
<jvms>
      <jvm name="Jvm1">
        <!-- le processes Jvm1 sera cree en utilisant la definition
      du processus nomme "localJVM" -->
            <creation>
                  <processReference refid="localJVM"/>
            </creation>
      </jvm>
      <jvm name="Jvm2">
         <!-- le processes Jvm2 sera cree en utilisant la definition
      du processus nomme "ssh_cluster" -->
            <creation>
                  <processReference refid="ssh_cluster"/>
            </creation>
      </jvm>
```

Le runtime référencé sous le nom `Jvm3` existe déjà, il n'est pas question ici d'en créer un nouveau mais de se connecter à l'existant. Les paramètres de connexion à ce runtime sont contenus dans l'entrée XML correspond au service d'acquisition des runtimes nommé `lookup_Jvm3`.

```
      <jvm name="Jvm3">
          <!-- Le runtime Jvm3 deja existe -->
            <acquisition>
                  <serviceReference refid="lookup_Jvm3"/>
            </acquisition>
      </jvm>
</jvms>
```

La dernière partie du descripteur décrit la manière de créer ou d'acquérir des runtimes.
Le processus `localJVM` va être créé sur la machine locale, la classe contenant la création du runtime est org.objectweb.proactive.core.process.JVMNodeProcess

```
<processDefinition id="localJVM">
      <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess">
      </jvmProcess>
</processDefinition>
```

Le processuss `sh_cluster` crée un runtime distant sur la machine `node1.cluster.inria.fr` en utilisant une connexion ssh. La manière de lancer le processus une fois connecté sur la machine distante sera la même que pour le processus `localJVM`.

```
<processDefinition id="ssh_cluster">
      <sshProcess class="org.objectweb.proactive.core.process.ssh.SSHProcess"
          hostname="node1.cluster.inria.fr">
             <processReference refid="localJVM"/>
      </sshProcess>
</processDefinition>
```

Le dernier processus consiste en l'acquisition du runtime "Jvm3" se trouvant sur la machine `predadab.inria.fr` et dont le nom est "PA_JVMsecureVM1". Dans cet exemple, le protocole de connexion à utiliser est `RMI`.

```
<services>
   <serviceDefinition id="lookup_Jvm3">
      <RMIRegistryLookup url="rmi://predadab.inria.fr/PA_JVMsecureVM1"/>
   </serviceDefinition>
</services>
```

Le concept des nœuds virtuels couplé avec celui des descripteurs de déploiement permettent la construction d'une application distribuée autour d'une architecture virtuelle. Selon ses besoins, un utilisateur peut ainsi modifier le déploiement de son application sans avoir à modifier le code source de celle-ci. Lors du déploiement, les ressources acquises peuvent l'être soit statiquement, en désignant explicitement la ressource dans le descripteur, soit dynamiquement en effectuant une demande de ressources auprès d'une base de données. Ainsi, lors de l'utilisation de grappes de calcul, deux exécutions successives d'une même application avec le même descripteur de déploiement peuvent ne pas représenter le même ensemble de machines. Après la phase de déploiement, le fichier de déploiement n'est plus utilisé jusqu'à la fin de l'exécution de l'application.

4.7 Conclusion

En conclusion, la philosophie exposée par la bibliothèque *ProActive* est de proposer un modèle de programmation distribué orienté objet dont l'intégration dans le code applicatif est la plus transparente possible. Cette orientation est mise en évidence, entre autre chose, par l'utilisation d'un protocole à méta-objets qui permet de découpler la gestion des concepts liés à la programmation distribuée de la logique de l'application. La configuration des méta-objets gérants les divers protocoles liés à la distribution de l'application se fait au moyen de descripteurs de déploiement favorisant ainsi l'adaptation du déploiement de l'application en fonction des solutions techniques disponibles. Il est ainsi possible de changer le protocole de communication, le système d'acquisition des ressources sans avoir à toucher au code de l'application.

Chapitre 5

Une architecture de sécurité de haut niveau

Les modèles présentés lors de l'état de l'art ont permis d'exposer un sous-ensemble représentatif des solutions existantes pour déployer et exécuter des applications dans des environnements distribués et sécurisés. Nous allons, dans un premier temps, exposer les hypothèses sur lesquelles nous basons notre mécanisme de sécurité ainsi que les objectifs recherchés en terme de sécurité. Le reste du chapitre nous permet d'introduire notre modèle de sécurité répondant à ces objectifs.

5.1 Hypothèses et objectifs

Le but de cette thèse est de fournir un modèle de sécurité pour des applications distribuées orientées grille de calcul. Nous prenons le terme d'application distribuée dans son sens le plus large, c'est-à-dire des processus s'exécutant sur divers ordinateurs pour mener à bien une tâche. Nous ne faisons pas de supposition sur le type des applications distribuées qui vont être déployées au-dessus de notre modèle.

Le postulat, fondamental, qui pose les bases de notre mécanisme de sécurité concerne la confiance accordée à chaque machine hôte sur laquelle s'exécutera le code de la bibliothèque.

Ce postulat nous donne les propriétés suivantes :

- le code de la bibliothèque sera exécuté correctement par les couches inférieures et notamment le système d'exploitation ;
- par transitivité, la confiance accordée à un hôte dépendra de la confiance accordée à l'identité associée au code de la bibliothèque ;
- la confiance en une entité s'exprime sous un format binaire : on a confiance ou non.

Les problèmes concernant la sécurité des applications réparties que nous abordons et pour lesquels nous amenons une solution sont :

- la protection des applications, des ressources et des utilisateurs les uns des autres. Autrement dit, on veut pouvoir contrôler l'accès à toutes ces entités ;
- la protection des communications échangées par les diverses parties en présence ;
- la préservation et le respect des politiques de sécurité de tous les objets d'une application distribuée et de celles des ressources disponibles qu'il s'agisse de ressources matérielles ou d'objets actifs.

Comme nous l'avons vu dans l'état de l'art et dans la présentation des intergiciels sécurisés, il existe des solutions de sécurité permettant de garantir la sécurité des applications distribuées. Ces solutions sont des concepts difficiles à maîtriser et cependant elles sont essentielles lors de la programmation d'applications distribuées. Elles imposent une modification du code de l'application, ce qui le rend moins lisible. On peut également noter que :

- ces modifications ne garantissent pas directement la correction du code de sécurité ajouté, ni celle de la politique de sécurité de l'application qui en résulte ;
- les besoins de l'application en termes de sécurité seront exprimés au sein du code de l'application et paramétrables seulement en modifiant le code source ;
- ce code devra être répliqué pour chaque application ayant les mêmes besoins.

La présentation de la librairie *ProActive* nous a permis de mettre en évidence les avantages d'implanter au sein de la librairie, le code de gestion de réservation et d'accès au ressources, la gestion des divers protocoles de communications, de migration, ...

Le premier but de cette thèse est de fournir un modèle de sécurité capable de gérer les vulnérabilités inhérentes aux applications distribuées selon les postulats énoncés précédemment. Afin de faciliter son utilisation, ce modèle doit être transparent au code métier de l'application, configurable en dehors de celui-ci.

Le deuxième postulat concerne la confiance du code du niveau méta de la bibliothèque. On supposera que ce code n'effectuera pas d'actions invalidant la politique de sécurité de l'application. Ce code évoluant au même niveau que le code gérant la sécurité, il lui serait possible d'outrepasser ses droits et d'effectuer des actions contraires à la politique de sécurité. On se place ici dans la même hypothèse que celle faîte par le standard EJB.

Lors du développement d'une application capable d'utiliser une grille de calcul, le développeur ne connaît pas la façon dont un utilisateur va déployer cette application. Il ne peut faire que des suppositions et donc il lui est difficile de coder le comportement en terme de sécurité de l'application. En effet, l'utilisateur peut aussi bien déployer l'application sur une seule machine d'une grappe de calcul, sur l'ensemble des machines de cette même grappe ou alors sur un ensemble de grappes de calcul (grille de calcul). Les besoins en terme de sécurité vont ainsi varier selon le déploiement de l'application. Par exemple, nous pouvons supposer que, lors d'un déploiement au sein d'une même grappe de calcul, seule l'authentification entre les diverses parties de l'application est nécessaire. Si cette même application utilise plusieurs grappes de calcul interconnectées par un réseau public, alors il est possible de vouloir simplement de l'authentification entre les nœuds d'une même grappe mais que lors de communications inter-grappes, la confidentialité et l'intégrité doivent être rajoutées.

Le deuxième but de cette thèse est ainsi de fournir un moyen à l'utilisateur de pouvoir configurer la sécurité de l'application selon le déploiement de cette dernière.

Finalement, les applications distribuées utilisant des grilles de calcul peuvent, au cours de leur exécution, acquérir dynamiquement de nouvelles ressources afin de s'y déployer pour augmenter la puissance de calcul à leur disposition. Cependant cette dynamicité ne doit pas compromettre la sécurité.

Le troisième but de cette thèse est d'intégrer cette notion de dynamicité des applications distribuées au sein du modèle de sécurité proposé afin de ne pas limiter la dynamicité de ce type d'applications. Cela impose de garantir la politique de sécurité de l'application tout au long de son exécution.

5.2 Un modèle de sécurité générique et hiérarchique

Le chapitre précédent nous a permis de décrire la bibliothèque *ProActive* et nous avons vu comment une application java s'exécute lorsque qu'elle est liée à la bibliothèque. À ce point, aucune notion de sécurité n'a été prise en compte.

Selon le principe du contrôle d'accès présenté dans la section 2.3, il faut, pour que le moniteur de référence puisse prendre une décision, pouvoir authentifier les deux parties en présence lors d'une interaction.

Si on reprend le modèle de base de *ProActive* on s'aperçoit que les entités qui peuvent interagir sont multiples, nous pouvons, dans un premier temps, identifier au moins les objets actifs et les runtimes. Cela signifie que nous allons devoir proposer un modèle de sécurité qui puisse prendre en compte à la fois la particularité des objets actifs en tant, par exemple, qu'entités mobiles mais aussi celles des runtimes en tant qu'entités capables de contenir d'autres entités comme les nœuds ou les objets actifs.

Par ailleurs, il est possible d'extraire des différents objets composant une architecture distribuée (runtimes, nœuds) et des objets actifs un ensemble distinct de fonctionnalités qui vont servir de base commune à la définition et la création d'une spécification concernant les besoins de sécurité d'une architecture distribuée. Cet ensemble de fonctionnalités est regroupé sous le terme d'*entité sécurisée* et nous définissons une entité sécurisée de la manière suivante :

Définition 10 *Entité Sécurisée*
Par définition, on nommera une entité sécurisée tout code ou donnée auquel est associé un mécanisme de sécurité permettant l'interception de toutes les interactions avec l'environnement extérieur. À une entité sécurisée sont associées une identité et une politique de sécurité. À partir de sa politique de sécurité, un gestionnaire de sécurité pourra déduire une politique de sécurité pour une interaction donnée.

Cette définition permet d'atteindre en partie l'un des objectifs de notre modèle : l'autonomie dont dispose l'entité sécurisée vis-à-vis de la prise de décision concernant les règles de sécurité. Du fait que nous nous trouvons dans une architecture répartie, aucun postulat ne peut être fait, par exemple, sur la connectivité du réseau sous-jacent ou sur tout autre défaillance pouvant survenir dans une architecture distribuée, le gestionnaire de sécurité doit pouvoir être autonome et pouvoir prendre les décisions concernant la sécurité de l'objet qu'elle sécurise de manière automatique sans avoir recours à un serveur de politique centralisé, ou en minimisant les communications vers des objets se trouvant à distance. En dotant l'entité sécurisée d'une politique de sécurité, le mécanisme de sécurité va pouvoir utiliser cette politique de sécurité pour prendre les décisions de sécurité nécessaires vis-à-vis de cette entité sécurisée.

Par ailleurs, on ne fait aucune supposition sur l'objet contenu par l'entité sécurisée. Notre modèle de sécurité n'est pas intrusif : il ne requiert aucune modification de l'objet sécurisé ni statiquement en modifiant le code source, ni dynamiquement par réflexivité. Cette approche permet l'utilisation et la sécurisation des objets java dont le code source n'est pas disponible mais aussi l'utilisation d'objets écris dans un autre langage (en C, C++, etc) et dont le chargement au sein de la machine virtuelle a été effectué par la Java Native Interface (JNI). De cette façon, l'objet sécurisé continue son exécution de manière classique, sans avoir nullement notion d'être sécurisé. Nous allons maintenant décrire plus en détails ce mécanisme d'interception.

La notion de hiérarchie qui va exister pendant l'exécution d'une application est importante du point de vue de la sécurité car elle induit qu'une entité sécurisée peut imposer une politique de sécurité précise aux entités qu'elle contient.

5.2.1 Contrôle d'accès aux entités

Afin de protéger un objet et étant donné qu'on ne peut pas modifier l'objet lui-même, il convient de sécuriser les échanges de flux entre cet objet et les objets extérieurs c'est-à-dire les appels de méthodes de et vers cet objet ainsi que les valeurs de retour résultantes de ces appels de méthodes. En d'autres termes, il faut intercepter tous les messages entrants et sortants.

La figure 5.1 expose la solution retenue. On place un objet d'interception, le gestionnaire de sécurité, entre les interactions venant du monde extérieur et l'objet protégé. Les messages entrant et sortant seront interceptés par le gestionnaire de sécurité. Au sein de l'entité les communications entre l'objet sécurisé et l'objet d'interposition s'effectuent en clair, tandis que les communications entre le gestionnaire de sécurité et les objets extérieurs à l'entité peuvent être

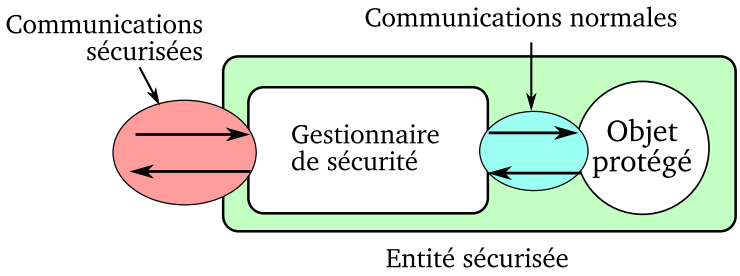


FIG. 5.1 – Protection d'un objet

sécurisées.

Étant donné que nous ne faisons pas d'hypothèse sur l'objet protégé, cet objet peut être également une entité sécurisée. Cette propriété induit la possibilité de mettre en place une structure hiérarchique, représentable sous la forme d'un arbre n-aire (figure 5.2), composée d'entités sécurisées. Du point de vue des politiques de sécurité, la structure arborescente permet de créer une politique de sécurité spécifique pour chaque entité sécurisée, l'héritage des politiques de sécurité permettant d'imposer automatiquement les politiques de sécurité d'une entité sécurisée parent à ses filles.
Il existe une forte analogie entre la structure hiérarchique des entités sécurisées et la structure hiérarchique présente au sein de la bibliothèque. Par exemple, pour un nœud contenant des objets actifs, si on superpose la structure des entités sécurisées sur celles du nœud et des objets actifs, on obtient un nœud sécurisé contenant des objets actifs sécurisés.

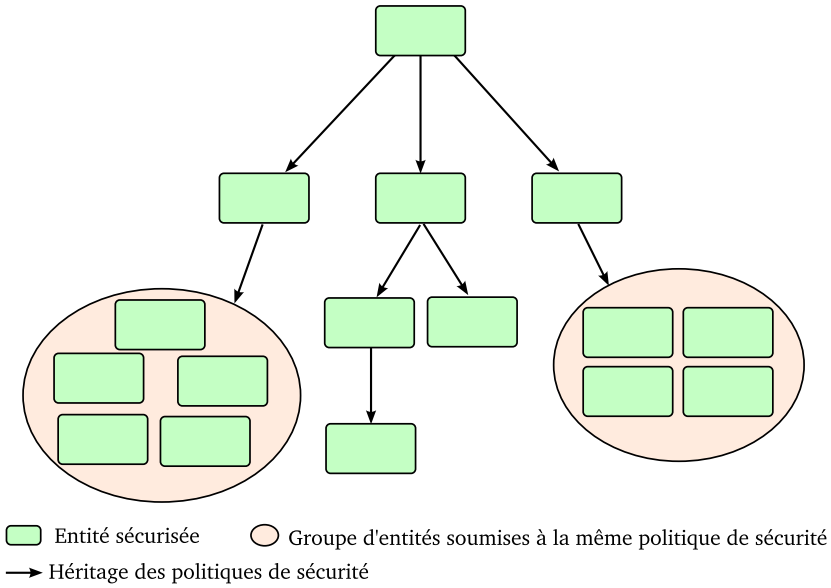
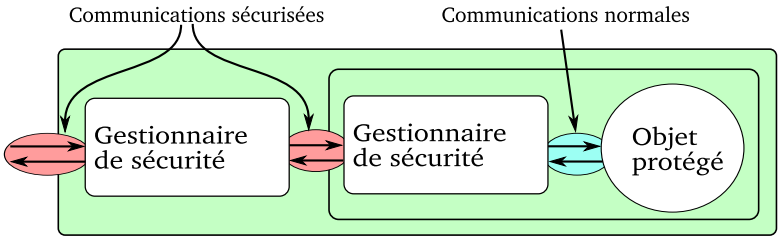


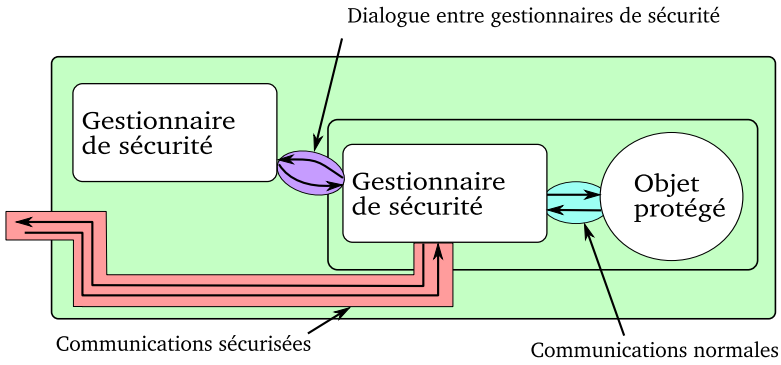
FIG. 5.2 – Combinaison arborescente des entités sécurisées

Cependant, la principale limitation de ce modèle réside dans la perte de performances au niveau des communications induite par la structure en arbre, chaque entité devant négocier une politique de sécurité et créer au besoin les objets cryptographiques nécessaires pour chiffrer la communication. Ainsi, si une entité sécurisée se trouve être une feuille de l'arbre, le chemin suivi par la communication passera par la racine de l'arbre et se propagera jusqu'à la feuille. Or à chaque nœud de l'arbre, le gestionnaire de sécurité établira une session sécurisée avec le ges-

tionnaire de l'entité fille (5.3, schéma a). Cette étape se répétera à chaque nœud jusqu'à ce que le message arrive à destination. Si on considère, que les politiques à appliquer à chaque com-



(a) Entité sécurisée sans optimisation



(b) Entité sécurisée avec optimisation

FIG. 5.3 – Hiérarchie d'entités sécurisées

munication imposent le chiffrement des communications, on se retrouve avec une succession de phases de chiffrement/déchiffrement du message. Cette redondance cryptographique induit non seulement une hausse de latence des communications, une forte baisse du débit des communications entre des objets actifs due au chiffrement et une surconsommation des ressources système en terme de mémoire et de temps CPU.
La structure arborescente pose un problème de performance au niveau de l'entité sécurisée correspondant à la racine de l'arbre car elle doit gérer toutes les communications de ses entités filles, elle agit comme un goulot d'étranglement. Cette racine étant le point d'entrée de toutes les communications, une panne de celle-ci impacterait toutes les entités filles.

Pour palier à ces problèmes, nous avons modifié la façon dont se comportent les entités sécurisées quand elles se trouvent emboîtées les unes dans les autres. Chaque entité possède une référence sur l'entité qui la contient (si elle existe). Les entités peuvent ainsi communiquer entre elles afin de convenir de la politique de sécurité à appliquer pour une communication donnée. C'est l'entité qui protège directement l'objet qui sera chargée d'appliquer la politique de sécurité résultant du calcul de toutes les politiques de sécurité s'appliquant à cette communication. Les autres entités n'ont comme seul rôle que de fournir la ou les politiques de sécurité qu'elles veulent imposer à cette interaction. De cette manière, l'entité sécurisée est complètement autonome dans la gestion de ses interactions avec le monde extérieur. Ses seules interactions avec ses entités parentes se font lors de l'initialisation d'une session. Nous obtenons par la même occasion, un chiffrement du message de bout-en-bout. Seule l'entité sécurisée destinataire du message aura la possibilité de déchiffrer ce dernier.

5.2.2 Le gestionnaire de sécurité

Une entité sécurisée est composée d'un gestionnaire de sécurité et d'un objet protégé. Le gestionnaire de sécurité est au cœur du système de sécurité d'une entité. Il sert de chef d'orchestre

et regroupe toutes les fonctions de sécurité nécessaires au bon fonctionnement du mécanisme de sécurité.

Il contient plusieurs objets aux fonctions bien distinctes comme :

- le *gestionnaire des sessions* qui contient les diverses sessions établies avec les autres entités. Ces sessions contiennent les différents paramètres de sécurité nécessaires à la communication avec une entité distante ;
- le *serveur de politique de sécurité* sert de base de donnée contenant les politiques de sécurité de l'entité. Ce sont ces politiques qui gouvernent l'établissement de sessions vers les autres entités ;
- le *serveur d'identité* contient les divers éléments permettant d'établir l'identité de l'entité, notamment son certificat publique, sa clé privée.

Une session représente une connexion établie ou en cours d'établissement entre deux entités. Une session contient :

- un identifiant : un numéro de session. Ce numéro sert à identifier une session existant entre deux entités ;
- un objet contenant la clé de session et ses paramètres d'initialisation ;
- une date d'expiration de la session ;
- l'identité de l'entité distante ;
- la politique de sécurité négociée entre les deux entités.

Pour une communication donnée entre deux entités, l'objet Session existe à la fois chez l'entité client mais aussi chez l'entité serveur, ces deux sessions comportent le même identifiant. Il est utilisé lors du transfert de messages chiffrés afin de retrouver la session et donc la clé permettant le déchiffrement du message.

5.2.3 Les entités sécurisées dans ProActive

L'entité sécurisée est notre élément de base sur lequel nous pouvons exprimer des politiques de sécurité. Cependant, la définition d'une entité sécurisée est une définition générique et il faut maintenant pouvoir tenir compte de la spécificité propre à chaque objet qui va être sécurisé. En effet, d'un point de vue technique, tous les objets partagent au minimum la possibilité de recevoir et d'émettre des appels de méthodes. Nous allons présenter les divers types d'entités sécurisées qu'il est possible de trouver au sein de la bibliothèque et leurs caractéristiques propres.

Les objets actifs sécurisés

Les premiers objets auxquels nous pensons lors de la sécurisation d'une application sont les composants de cette application : les objets actifs. Un objet actif peut effectuer des appels de méthode sur d'autres objets actifs ou sur des runtimes ou en recevoir comme tout objet Java normal. Cependant il possède aussi des comportements propres comme, par exemple, la possibilité de migrer d'un nœud vers un autre. L'entité sécurisée dans son comportement initial n'est plus suffisante. A la simple interception des requêtes et des réponses, il faut maintenant ajouter la gestion de la mobilité d'un objet actif. Il faut non seulement pouvoir spécifier des règles de sécurité concernant la migration mais aussi introduire un algorithme permettant de gérer à la fois la migration et la sécurisation d'un objet actif. Nous présentons et discutons le modèle retenu dans la partie 6.3 de ce manuscrit.

Les nœuds sécurisés

Initialement les nœuds servent à nommer les divers lieux d'exécution disponibles qu'ils soient locaux ou distants et à accueillir les objets actifs. Notre approche étend cette fonctionnalité initiale en lui ajoutant les notions de sécurité nécessaires. Ainsi d'un simple conteneur passif d'objets actifs, le nœud devient un acteur du mécanisme de sécurité de la bibliothèque. Il est capable pour chacune de ses fonctionnalités de rechercher au sein de sa politique de sécurité si l'action entreprise par l'entité sécurisée tierce peut être autorisée ou non ainsi que les attributs de sécurité à utiliser pour le chiffrement des communications.

En tant qu'entité sécurisée, un nœud est capable d'imposer sa politique de sécurité aux entités sécurisées qu'il contient à savoir les objets actifs. Ainsi que nous l'avons vu lors de la présentation générale de la bibliothèque (chapitre 4), un nœud dépend du déploiement d'une application donnée, aussi la politique de sécurité d'un nœud va dépendre de la politique de sécurité de l'application à laquelle il appartient. Cette fonctionnalité est exposée dans la section 5.6 de ce chapitre.

Les runtimes sécurisés

Un runtime représente un environnement d'exécution pour les applications. Il accueille les processus utilisateurs (i.e. objets actifs) et leur fournit les ressources locales nécessaires à leur fonctionnement. Dans sa version sécurisée, un runtime peut être associé à une politique de sécurité afin de contrôler, d'une part, les services qu'il offre aux applications locales ou distantes, et d'autre part, le comportement des objets actifs s'exécutant à l'intérieur du runtime. Parmi les services distants offerts, nous pouvons lister les principaux tels que la *création* d'un nœud ou d'un objet actif, la *réception* d'un objet actif migrant vers le runtime ou la *création* d'un nouveau runtime. Le runtime étant un des points d'entrée sur la machine, il est important de sécuriser tous ces services.

Dans le cadre d'applications collaboratives, un runtime lancé par un utilisateur peut accueillir un ou plusieurs objets actifs appartenant à un autre utilisateur (une autre identité). Par extension, cela peut aussi servir de base pour offrir, de manière sécurisée, un environnement d'accueil pour des agents mobiles.

Nous avons présenté les évolutions des concepts déjà existants vers une version sécurisée. Actuellement, l'entité qui se trouve la plus haute dans la hiérarchie est le runtime. Au niveau de la machine qui l'héberge, un runtime est vu comme une application locale s'exécutant sur une seule machine. La politique de sécurité initiale d'un runtime est définie lors du démarrage du runtime. Sur une machine donnée, plusieurs runtimes lancés par des utilisateurs différents et avec des politiques de sécurité différentes peuvent s'exécuter en parallèle. De plus, il faut noter que la bibliothèque est prévue pour être utilisée sur la plupart des configurations matérielles existantes. Or parmi ces configurations, certaines possèdent des caractéristiques communes et spécifiques, comme toutes celles qui peuvent rentrer dans la définition d'une grille de calcul, à savoir un ensemble de machines qui, du point de vue de la sécurité doivent être soumises aux mêmes politiques de sécurité. En effet, le démarrage d'un runtime peut être effectué par toute personne qui possède un accès à une machine donnée. Cependant, l'administrateur d'un parc de machines peut vouloir imposer une politique de sécurité commune à tous ces ordinateurs tout en ne souhaitant pas laisser le soin aux utilisateurs de positionner eux-mêmes cette politique de sécurité propre au parc de machines lors du démarrage de leurs runtimes. Par souci de simplicité mais aussi de sécurité, il est important de séparer les divers niveaux de politiques de sécurité et de faire en sorte que chaque acteur ne s'occupe que de définir la politique de sécurité propre à son niveau. La solution que nous proposons est l'utilisation de *Domaines de sécurité*.

5.2.4 Les domaines de sécurité

L'organisation en domaines de sécurité est une manière standard de structurer (virtuellement) les organisations qui participent à une grille de calcul. Les domaines sont les entités logiques permettant d'exprimer des politiques de sécurité de manière hiérarchique.
La définition que nous utiliserons dans notre approche est la suivante :

Définition 11 *Domaines de sécurité*
Un domaine est une entité logique regroupant un ensemble d'entités. Ces entités contenues peuvent être elles-mêmes des domaines (on parle alors de sous-domaines) ou des runtimes. Par extension, un domaine sécurisé est une entité sécurisée. Il est ainsi possible d'associer une politique de sécurité à un domaine.

Un domaine sert de serveur de base de données de politiques de sécurité. Cependant, il est possible de définir dans la politique de sécurité d'un domaine des règles s'appliquant aux entités

sécurisées contenues dans le domaine. Par exemple, si un administrateur définit, au niveau d'un domaine de sécurité, une règle interdisant la migration des objets actifs, cette règle sera imposée à toutes les entités sécurisées contenues dans le domaine. Ainsi, quelles que soient les règles définies dans les entités contenues, aussi bien au niveau d'un runtime, que d'un nœud ou de l'objet actif lui-même, la migration de l'objet actif sera interdite par le mécanisme de sécurité.

Ces deux entités, les domaines et les runtimes, permettent la création d'une infrastructure sur laquelle des applications vont pouvoir s'exécuter. En ce qui concerne les domaines, les politiques de sécurité sont gérées par les administrateurs en charge du domaine avec la possibilité de subdiviser le domaine en sous-domaines pour faciliter leur gestion. Concernant les runtimes, les politiques de sécurité sont définies par les personnes pouvant activer ces runtimes, il s'agit généralement d'utilisateurs possédant un compte sur la machine et qui veulent offrir/vendre/utiliser des ressources comme la puissance de calcul ou des espaces de stockage. L'identité d'un runtime dépend de l'identité de l'utilisateur qui l'a démarré. De ce fait, on limite les possibilités de corruptions physiques de la machine au compte utilisateur qui héberge les runtimes. Contrairement aux nœuds ou aux objets actifs dont la politique de sécurité peut être créée dynamiquement, les politiques de sécurité des domaines et des runtimes sont mises en place de manière *statique* soit par un administrateur, soit par un utilisateur.

Nous avons présenté les diverses entités sécurisées induites par le modèle de la bibliothèque (objets actifs, nœuds, runtimes) et celles dont l'architecture distribuée impliquait la création (les domaines). Pour chacune de ces entités, nous avons exposé ses principales fonctionnalités et introduit les interactions avec le mécanisme de sécurité. Nous allons maintenant présenter un classement des divers types de politiques de sécurité induit par la structure hiérarchique de notre modèle.

5.2.5 Taxonomie des politiques de sécurité

Tous les différents niveaux de politiques de sécurité peuvent être classés selon trois grandes catégories représentées au sein de la figure 5.4.

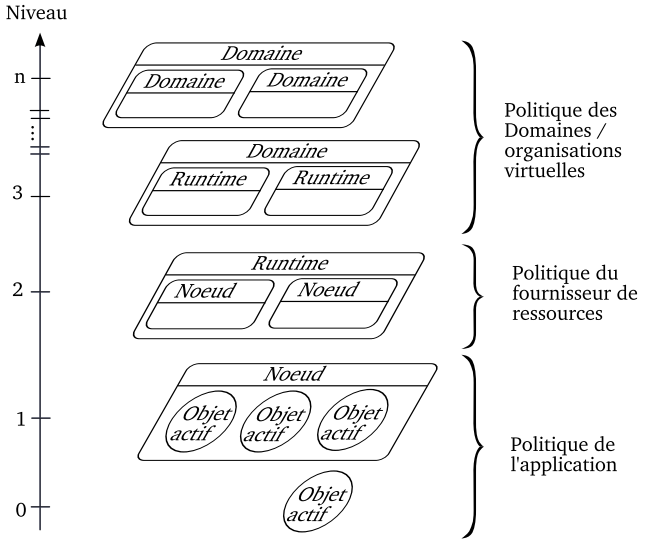


FIG. 5.4 – Les différentes hiérarchies de sécurité

La première catégorie correspond aux niveaux les plus bas de la structure arborescente et elle englobe à la fois les entités sécurisées représentant les objets actifs et les nœuds. Cette catégorie est appelée *Politique de l'application*. Les politiques de sécurité de cette catégorie sont fortement liées à une exécution d'une application donnée. En effet, les objets actifs et les nœuds sont des objets spécifiques à une exécution d'une application donnée. Cette catégorie de politique

de sécurité dépend essentiellement de la façon dont l'application va être déployée. C'est sur cette catégorie que l'utilisateur final peut agir. Lors du déploiement de son application, il va spécifier une politique de sécurité. Cette politique de sécurité sera répliquée sur tous les objets composant l'application.

La deuxième catégorie concerne spécifiquement les runtimes. Si on regarde la définition d'un runtime, il fournit des services et les applications les utilisent. C'est pourquoi cette catégorie de politiques de sécurité est nommée *Politique du fournisseur*. Lorsqu'une application utilise un runtime, il peut :

- soit avoir été créé spécifiquement pour l'application ; dans ce cas, sa politique de sécurité sera généralement la même que celle de l'application dont il dépend ;
- soit exister en dehors du contexte de l'application et fournir les ressources nécessaires aux applications autorisées ; dans ce cas, sa politique de sécurité dépend de paramètres extérieurs.

Dernière catégorie, la *Politique des domaines* sert essentiellement à la structuration logique d'organisations virtuelles et au regroupement sous un même domaine des diverses ressources mises à la disposition des utilisateurs et/ou des fournisseurs de ressources. Elle est essentiellement réservée aux administrateurs des machines sur lesquelles les runtimes s'exécutent.

5.2.6 Conclusion

L'approche hiérarchique que nous avons présentée possède deux avantages primordiaux :

- premièrement, elle permet une gestion décentralisée dans la création des politiques de sécurité. Ce n'est plus à un seul acteur de définir la politique globale d'un site et d'une application mais à différents acteurs. Un acteur (administrateur d'un site, fournisseur de ressources, utilisateur final) du niveau n définit sa propre politique de sécurité. Cette politique sera imposée à tous les acteurs des niveaux inférieurs ($n-1$ à 0).
- deuxièmement, cela implique que, pour une interaction donnée, ce n'est pas une seule politique de sécurité qui va s'appliquer mais un ensemble de politiques de sécurité.

De plus, afin de gérer la décentralisation et la distribution des éléments sécurisés de l'application, nous avons doté chaque élément sécurisé d'une autonomie dans la prise de décisions de sécurité grâce à la structuration en entités sécurisées.

Ainsi l'entité sécurisée est le concept fondamental utilisé au sein du modèle de sécurité présenté dans cette thèse. Maintenant que nous avons une vue d'ensemble de l'architecture de sécurité, nous pouvons présenter les mécanismes introduits pour gérer l'identification des entités sécurisées, la manière d'exprimer des politiques de sécurité et comment, à partir d'un ensemble de politiques de sécurité, les combiner entre elles pour n'en former qu'une seule qui sera appliquée à une interaction donnée.

5.3 Authentification des entités

Dans cette section, nous allons nous intéresser à l'authentification des entités sécurisées. Lors d'une interaction, le gestionnaire de sécurité doit pouvoir identifier les entités sécurisées impliquées afin de rechercher les règles de sécurité concernant cette interaction.

Il faut se rappeler que l'un des buts poursuivis dans notre approche est de proposer une gestion de la sécurité la plus transparente et configurable possible aussi bien pour l'utilisateur que pour le développeur. Pour cela, on doit au mieux supprimer ou, dans le pire des cas, minimiser une interaction entre le code de sécurité et l'utilisateur. Il nous faut une solution technique qui permette l'automatisation des divers processus entrant en jeu dans la gestion de la sécurité. Dans le cas qui nous intéresse maintenant, notre solution doit pouvoir créer les objets nécessaires à l'identification d'une entité automatiquement.

En nous basant sur l'étude bibliographique présentée dans ce manuscrit et des contraintes suscitées, la solution d'authentification qui s'impose est l'utilisation d'une architecture à clé publique de type SPKI. Le choix de la solution SPKI s'explique par la nature même des entités à identifier. En effet, l'infrastructure SPKI permet l'attribution de certificats à des objets logiques comme les runtimes, les nœuds ou encore les objets actifs ce qui n'est pas le cas de l'infrastructure PKI dont les certificats identifient une entité physique ou morale en donnant son nom, son émail, sa localisation, ... Ainsi , l'identification des diverses entités sécurisées se fera en attribuant une identité unique à chaque entité au moyen de certificats.

Une autre fonctionnalité intéressante réside dans la possibilité d'établir une chaîne de certificats. Le chaînage de certificats permet d'obtenir l'identité de toutes les entités qui ont successivement généré des certificats jusqu'au certificat final qui identifie l'entité courante. En d'autres termes, à partir du certificat d'une entité, nous sommes capables de retrouver le certificat de l'utilisateur qui l'a lancée. Il n'existe aucune limitation à la taille de cette chaîne de certificats. Cependant il convient de modérer la taille de cette chaîne : étant donné que la validation d'un certificat requiert la validation de tous les certificats présents dans la chaîne, plus cette chaîne est courte, plus la validation du certificat sera rapide.

5.3.1 Un schéma d'authentification hiérarchique

Le principal problème auquel nous devons faire face réside dans la dynamicité des applications distribuées. En effet, une application évolue tout au long de son exécution. Son nombre d'objets ou le nombre de nœuds sur lesquels elle est déployée, par exemple, peut augmenter et diminuer dynamiquement suivant plusieurs paramètres externes ou internes à l'application. Du point de vue de la sécurité, on doit pouvoir générer des certificats pour ces nouvelles entités sécurisées sans pour autant devoir interagir avec l'utilisateur.

Selon le principe même des certificats, chaque certificat engendré doit être unique dans un contexte et représenter une entité sécurisée donnée. Il est certes possible d'écrire le code de création du certificat et d'effectuer l'association du certificat et de l'entité au sein de l'application. Cependant nous recherchons à automatiser ce processus et à le rendre transparent aux développeurs et aux utilisateurs.

Afin de mettre en évidence l'inadéquation du modèle standard des certificats et le besoin d'adaptation de ce mécanisme dans le cadre de notre modèle de sécurité, nous utiliserons le scénario suivant :

1. L'utilisateur, muni de son certificat fourni par l'autorité de certification, crée l'objet appelé `Entité 1`. En tant qu'entité sécurisée, cet objet reçoit un certificat.
2. `Entité 1` va ensuite créer les entités `Entité 2` et `Entité 3`.
3. `Entité 3` crée un nouvel objet actif `Entité 4`,
4. finalement, `Entité 4`, à son tour, crée un objet actif `Entité 5`.

En premier lieu, nous présentons la façon dont s'organise le chaînage de certificats dans le cas standard (figure 5.5). Un certificat comporte une partie référençant l'entité qui l'a généré et une partie qui l'identifie. Dans notre exemple, chaque certificat, hormis celui de l'utilisateur, représente une entité sécurisée.
Le scénario dans le cas standard est le suivant :

1. Le certificat de l'`Entité 1` contient l'identité de l'utilisateur X dans la partie identifiant le créateur du certificat.
2. Le certificat de l'`Entité 2` et de l'`Entité 3` contiennent l'`Entité 1` en tant que créateur du certificat.
3. Le certificat de l'`Entité 4` contient l'`Entité 3` en tant que créateur du certificat.
4. Le certificat de l'`Entité 5` contient l'`Entité 4` en tant que créateur du certificat.

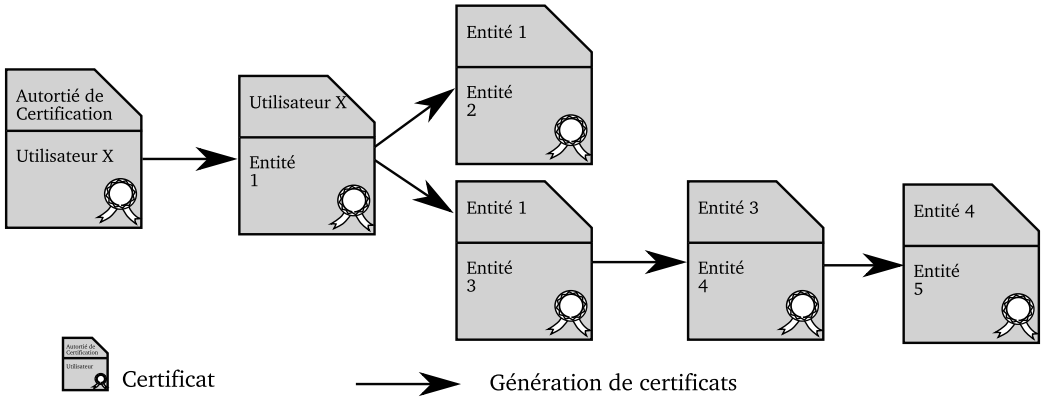


FIG. 5.5 – Chaînage de certificats : approche standard

Si on s'intéresse aux chaînes de certificat générées, on se retrouve avec des chaînes de certification de longueurs différentes. Dans le cadre d'une opération effectuée par l'`entité 5`, il va nous falloir valider quatre certificats afin de retrouver l'utilisateur qui a lancé l'application.

Un autre problème se pose avec cette approche, la *séparation des contextes*. Quand un utilisateur lance plusieurs applications qu'il s'agisse de plusieurs instances de la même application ou d'applications différentes, il ne veut pas nécessairement que les entités des diverses applications puissent communiquer entre elles. Or, si on s'intéresse à la figure 5.5, on s'aperçoit qu'il est possible que les entités 1,3,4,5 appartiennent à la même application étant donné le chaînage de certificat existant. Il n'est cependant pas possible de déduire quelque chose pour les entités 2 et 3. La seule information que nous apporte ce schéma est qu'elles appartiennent toutes les deux à l'*utilisateur X*. Il ne nous est pas possible d'affirmer que ces deux entités appartiennent à la même application ou pas.

Pour toutes les raisons exposées, nous avons introduit un nouveau schéma d'identification des entités afin d'ajouter un *contexte d'application* et ainsi pouvoir identifier à quelle application appartient une entité donnée. Notre approche reprend le principe du chaînage de certificats mais elle introduit une variation dans son utilisation. Nous allons ajouter au sein de la chaîne des certificats un certificat intermédiaire qui identifiera l'application. Ce certificat porte le nom de *certificat d'application*. D'un point de vue technique, chaque entité doit à la fois posséder son propre certificat mais aussi celui de l'application dont elle dépend. Lors de la création d'une nouvelle entité, cette dernière recevra en plus de son certificat, celui de l'application dont elle dépend. Elle sera ainsi capable de créer, à son tour, des certificats pour les nouvelles entités qu'elle instanciera. La figure 5.6 présente cette nouvelle chaîne de certificats. Cette approche nous permet de garantir la séparation des contextes d'exécution d'une même application. Par définition, on considérera que deux entités appartiennent à la même application si et seulement si elles dérivent du même certificat d'application.

La figure 5.7 reprend le scénario présenté précédemment et présente la nouvelle chaîne de certificats obtenue en utilisant notre modèle. À partir du certificat d'une entité, il est possible d'obtenir le certificat de l'application à laquelle elle appartient et de comparer des certificats entre eux afin de savoir s'ils appartiennent à la même application.

5.3.2 Avantages et inconvénients de l'approche

Nous avons fait le choix de doter chaque entité sécurisé d'un certificat propre. Le principal inconvénient de notre approche réside dans le temps nécessaire à la génération de la paire de clés asymétriques nécessaire lors de la création d'un certificat. Ainsi une application qui passe son temps à créer des objets actifs se trouvera pénalisée dans son fonctionnement. Cependant, cette remarque s'applique également, dans une moindre mesure, au concept d'objet actif sans sécurité.

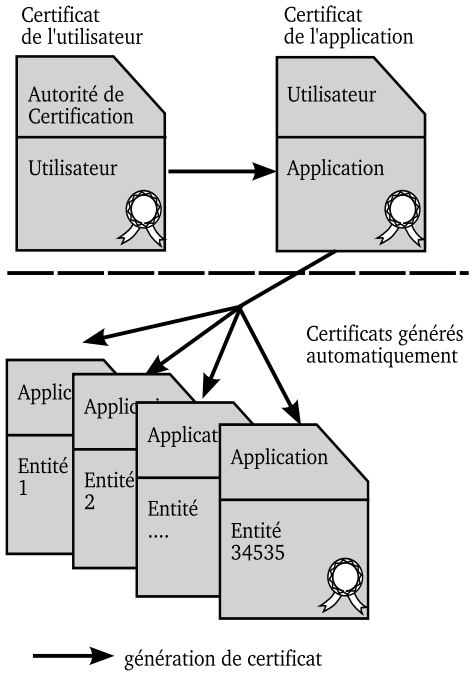


FIG. 5.6 – Authentification avec utilisation d'un certificat d'application

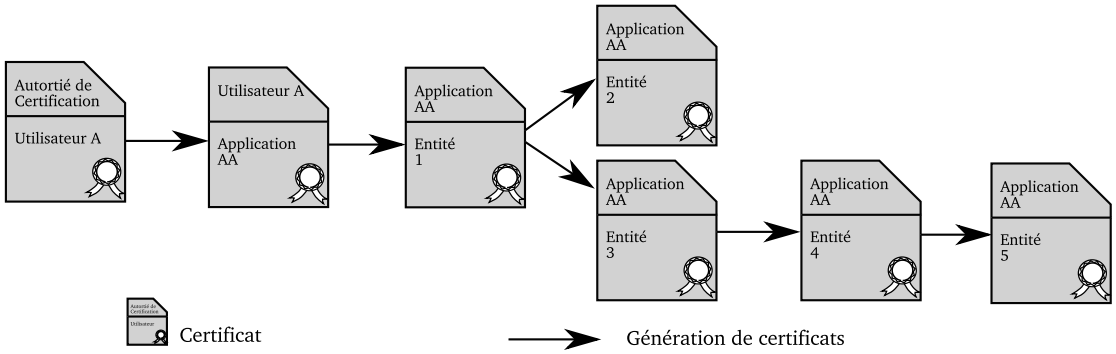


FIG. 5.7 – Chaînage de certificats : version avec un certificat d'application

Un objet actif est, comme nous l'avons décrit dans la section 4.2 un objet de grain moyen, composé de plusieurs méta-objets et d'un fil d'exécution. Sa création est elle-même coûteuse par rapport à la création d'un objet java standard. Il nous semble raisonnable de supposer que lorsqu'un objet actif (sécurisé ou non) est créé, sa durée de vie est largement supérieure au temps de sa création.

Le premier avantage de notre modèle réside dans son adaptation avec la définition générique d'une entité sécurisée que nous avons donnée. Une approche alternative que nous avions envisagée était d'utiliser le certificat d'application pour toutes les entités de l'application, la distinction entre les entités pouvant être réalisée en leur associant à chacune un nom symbolique. L'inconvénient de cette approche est la création de deux groupes d'entités distinctes : celles possédant leur propre certificat comme les domaines et les *runtimes* et celles représentant des objets de l'application comme les nœuds et les objets actifs. Dans ce cas, nous perdions donc la définition générique d'une entité sécurisé qui est identifié par un certificat unique.

Le fait d'associer un certificat à une entité permet de l'identifier en dehors de tout contexte, ce qui nous sera utile lors de l'écriture de politiques de sécurité génériques.

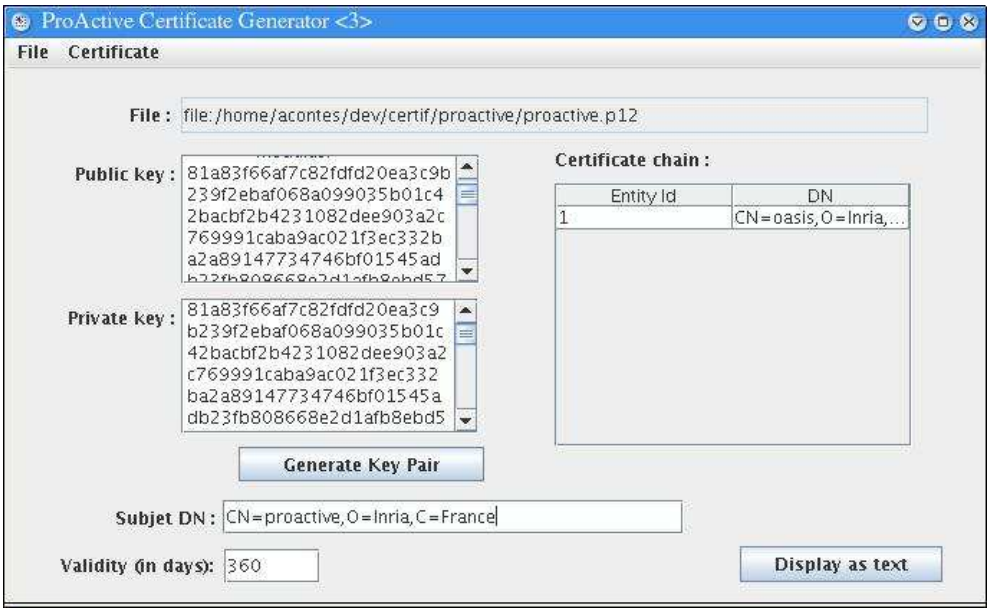


FIG. 5.8 – Outil de gestion des certificats

Chaque entité ayant un certificat différent, cela rendra la cryptanalyse des communications de l'application beaucoup plus difficile. Si on suppose qu'un attaquant a réussi à calculer la clé privée d'une entité, ceci ne remet en cause que les communications de cette entité et non toutes les communications de l'application.

Le processus d'authentification dans une infrastructure à clé publique implique la vérification des certificats ; il faut à la fois vérifier que le chaînage de certificat est correct mais également que le certificat n'a pas été révoqué. L'approche SPKI suppose que le principal ayant généré un certificat soit toujours accessible afin que le processus de vérification puisse s'assurer que le certificat n'a pas été révoqué. Dans notre cas, cela suppose une machine accessible par tous afin de répondre aux requêtes de validation. Le service de sécurité de Globus est confronté aux mêmes problèmes [121] concernant les certificats *proxy certificates* utilisés pour la délégation dynamique. Leur approche consiste à créer des certificats possédant une durée de vie courte (huit heures) et à vérifier seulement la validité de la chaîne de certification. Lorsqu'un de ces certificats arrive en fin de vie, et qu'une action sécurisée est nécessaire, le service de sécurité demande la génération d'un nouveau certificat au principal ayant créé le certificat arrivant en fin de vie. Cela suppose tout comme dans SPKI que le principal est toujours accessible.

Notre approche actuelle ne supporte pas le remplacement de certificats comme dans Globus. L'authentification est basée sur le chaînage de certificats et la durée de validité du certificat. Un certificat sera validé si on trouve dans la chaîne de certification un certificat autorisé par la politique de sécurité. Il faut bien évidemment que la chaîne de certification soit valide et que le certificat ne soit pas périmé.

5.3.3 Génération des certificats

Nous avons créé une interface graphique regroupant les fonctionnalités nécessaires à la gestion des certificats. Elle permet la création de certificats au format PKCS#12, l'importation et l'exportation de certificats vers d'autres formats standard pouvant être utilisés par d'autres logiciels. Il est possible d'utiliser l'interface graphique pour générer des certificats pour tous les types d'entités ainsi que des certificats d'application. La figure 5.8 présente une entité dont le certificat contient une chaîne de certificats.

5.3.4 Conclusion

Dans cette partie, nous avons présenté le modèle d'authentification qui a été créé afin d'identifier les diverses entités de notre système. L'introduction d'un certificat d'application permet la différenciation des diverses applications lancées par un utilisateur et ainsi de créer un contexte de sécurité qui sera propagé à tous les membres d'une même application.

Le chaînage de certificats induit une composition hiérarchique des certificats qui, comme nous allons le montrer dans la suite du manuscrit, autorise une grande souplesse dans l'écriture de règles de sécurité.

5.4 Politiques de sécurité décentralisées et adaptatives

Nous nous intéressons ici à la manière d'exprimer les règles de sécurité qui vont gouverner le comportement des entités sécurisées.

5.4.1 Vers une politique de contrôle d'accès discrétionnaire

La partie 2.6 de ce manuscrit nous a permis d'introduire les divers types de politiques de sécurité existants. Notre choix s'est porté sur la politique de contrôle d'accès discrétionnaire pour plusieurs raisons. Tout d'abord, ce type de politique est le plus simple à mettre en œuvre, nous devons garder à l'esprit que les règles de sécurité peuvent être écrites par des utilisateurs qui ne maîtrisent pas forcément tous les concepts ni toutes les terminologies propres à la sécurité.

Ensuite, en accord avec notre modèle, la création puis la gestion par le mécanisme de sécurité des politiques sont réalisées de manière décentralisée. Chaque acteur pouvant écrire sa propre politique de sécurité sans avoir à interagir avec un autre acteur, il est ainsi inapproprié d'utiliser une politique de sécurité à base de rôles qui suppose une gestion soit coordonnée, soit centralisée, de la définition des divers rôles existants. La même remarque peut être utilisée concernant le contrôle d'accès obligatoire car il se base sur des niveaux d'autorisation dont la gestion doit être soit centralisée, soit coordonnée entres les divers sites participants.

5.4.2 Anatomie d'une politique de sécurité

Une politique de sécurité nous permet d'exprimer les conditions de sécurité à appliquer pour un cas précis. Pour cela, elle doit nous permettre, tout d'abord, d'identifier un ensemble d'entités sécurisées représentant l'ensemble des sujets initiant l'interaction et un autre ensemble d'entités sécurisées représentant l'ensemble des entités cibles de l'interaction. Une fois les différents ensembles clairement identifiés, il faut pouvoir exprimer les règles de sécurité qui s'appliqueront sur cette interaction.

La syntaxe des règles de sécurité a tout d'abord été définie au sein d'un langage créé pour nous permettre d'écrire simplement les règles de sécurité. La grammaire de ce langage est donnée en annexe A. La bibliothèque ayant évolué vers l'utilisation du XML pour ses fichiers de configuration et par soucis de cohérence, ce langage a été par la suite remplacé par un `schema` qui a permis de conserver l'expressivité de notre langage tout en étant au format XML. Cependant la plupart des articles que nous avons écrit reprennent le langage créé initialement car il a l'avantage d'être plus concis et lisible que la syntaxe XML.

Chaque politique de sécurité se décompose en trois parties bien précises comme le montre la figure 5.9. La première partie permet de définir les identités des divers acteurs (entités sources et entités cibles) de la politique. La deuxième partie décrit les interactions sur laquelle la politique de sécurité va imposer des restrictions. La troisième partie concerne les échanges de flux de données et la façon de les transmettre sur le réseau.

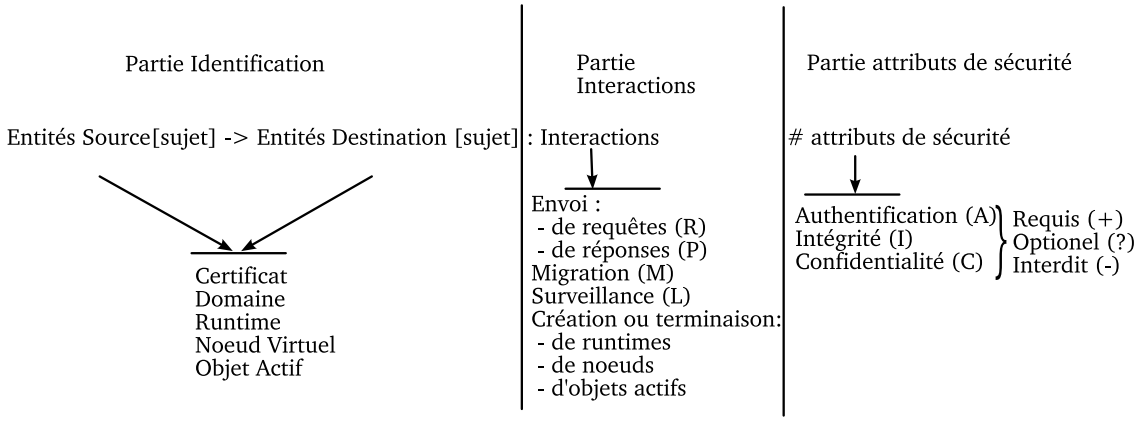


FIG. 5.9 – Syntaxe d'une règle

Identifications

La partie *identification* est la première partie d'une règle de sécurité. Elle est elle-même subdivisée en deux parties :

- la première partie contient l'ensemble d'entités à partir desquelles l'interaction a été initiée. Elles sont contenues dans la balise `<From>`.
- La deuxième partie représente l'ensemble d'entités cibles de l'interaction. Elles sont contenues dans la balise `<To>`.

L'entrée XML correspondant à une entité est la suivante :

```
<Entity type="file" name="arnaud.pkcs12"/>
```

La balise `<Entity>` comporte deux attributs `type` et `name`. Le contenu de `name` dépend directement de la valeur contenue dans `type`. Le tableau 5.1 présente les différentes valeurs possibles et leur description.

On parle ici d'ensembles d'entités à la fois pour les entités appelantes et pour les entités appelées car il est possible de spécifier plusieurs entités dans chacune de ces sous parties. Chaque ensemble correspond à une liste non ordonnée d'entités. Lors de l'évaluation de la liste, l'ensemble est interprété comme l'intersection de tous les éléments de cet ensemble. Cela signifie que pour qu'un ensemble soit considéré comme valide pour une interaction donnée, tous les éléments de cet ensemble doivent être présents dans l'ensemble des entités constitué dynamiquement par le gestionnaire de sécurité lors du début de l'interaction.

Un ensemble d'entités sécurisé s'écrit en concaténant les diverses entités les unes à la suite des autres. La ligne XML ci-après spécifie que, pour que la règle soit considérée comme valide, il faut qu'à la fois l'entité appelante se trouve dans un nœud issu du nœud virtuel `VN_1` et dans le runtime `Runtime_Cluster_Sophia`. Étant donné que nous utilisons des noms symboliques à la fois pour le runtime et pour le nœud, ces deux entités doivent faire partie de la même application donc dériver du même certificat d'application que l'entité appelante.

```
<From>
 <Entity type="VirtualNode" name="VN_1"/>
 <Entity type="Runtime" name="Runtime_Cluster_Sophia"/>
</From>
<To>
 <Entity type="VirtualNode" name="VN_1"/>
</To>
```

En pratique, remplir un des ensembles avec deux entités revient à dire que, pour que cette partie de la règle soit validée, les deux entités présentes dans la règle doivent se trouver dans l'ensemble des entités appelantes. Ainsi, il est possible de spécifier qu'une règle ne sera valide

Contenu de l'attribut type	Description de l'attribut name
File	name contient un chemin vers un fichier encodé au format pkcs12. Ce fichier contient à la fois le certificat de l'entité qu'il représente et la chaîne des certificats dont le certificat est issu. Les certificats que nous pouvons retrouver ici sont ceux d'entités dont le certificat ne va pas être changé tout le temps. Parmi ces entités, on peut retrouver les domaines, les utilisateurs et les runtimes. `<Entity type="file" name="arnaud.pkcs12"/>`
VirtualNode	l'attribut name contient le nom d'un nœud virtuel qui existe aussi dans le descripteur de déploiement. Cette entité représente tous les nœuds réels qui vont être créés à partir du nœud virtuel lors de l'activation du descripteur de déploiement. `<Entity type="VirtualNode" name="VN1"/>`
ApplicationCertificate	il s'agit d'une entité virtuelle représentant le certificat de l'application qui n'est généralement pas connu avant le déploiement réel de l'application. `<Entity type="ApplicationCertificate"/>`
Runtime	l'attribut name contient le nom d'un runtime qui existe dans le descripteur de déploiement. `<Entity type="Runtime" name="Runtime1"/>`
DefaultVirtualNode	Cette entité permet de spécifier un comportement valable sur n'importe quel nœud de l'application
Default	Cette entité représente n'importe quelle entité. Elle permet de spécifier des comportements par défaut. Elle peut être utilisée aussi bien comme entité source que comme entité cible. On peut aussi concevoir une règle qui possède comme entité source et comme entité cible des entités par défaut. Dans ce cas, le contenu de l'attribut name est ignoré. `<Entity type="Default" name=""/>`

TAB. 5.1 – Les différents types d'entités

que dans un nœud donné (première entité) d'un runtime donné (deuxième entité). Parallèlement, on retrouve les mêmes possibilités de filtrage au niveau de l'ensemble des entités représentant les entités appelées.

La structure hiérarchique introduite par le chaînage des certificats permet d'ajouter une fonctionnalité essentielle dans la conception de règles génériques. Si on considère la chaîne de certificats suivante `Organisation1 -> Utilisateur1 -> Application1`, où chaque entrée correspond à un certificat, le certificat `Utilisateur1` a été créé à partir du certificat `Organisation1`. On considère alors que `Utilisateur1` appartient à `Organisation1`, soit

$$\texttt{Application1} \subset \texttt{Utilisateur1} \subset \texttt{Organisation1}$$

et que toute règle impliquant `Organisation1` s'appliquera à `Utilisateur1`. Ainsi, il est possible d'écrire la règle suivante :

```
<From>
 <Entity type="File" name="organisation1.pkcs12"/>
</From>
<To>
 <Entity type="Default" name=""/>
</To>
```

Cette règle s'appliquera à toutes les applications lancées par l'utilisateur `Utilisateur1` quand elles voudront établir une interaction vers n'importe quelle autre entité.

Interactions et Attributs de sécurité

Les politiques de sécurité doivent être capables de contrôler toutes les *interactions* qui peuvent survenir lors du déploiement et de l'exécution d'une application multi-utilisateurs sur une grille de calcul. Les interactions couvrent les actions allant de la création de processus (JVM) à la surveillance des activités (objets actifs) à l'intérieur des processus et incluent évidemment la gestion des communications. Le tableau 5.2 dresse la liste des interactions gérées par le mécanisme de sécurité.

Au sein de la bibliothèque, les interactions correspondent à des appels de méthodes réalisés, soit vers des méta-objets spécifiques du body local, soit sur le body de l'objet distant. Une interaction se résume du point de vue des communications à un appel de méthode spécifique qui peut générer des échanges de messages. Il convient de pouvoir protéger ces messages en spécifiant les attributs de sécurité qui seront attribués au message. Le code XML gérant cette partie d'une règle de sécurité est présenté dans le listing suivant :

```
<Communication>
 <Outgoing value="authorized">
   <Attributes authentication="required" integrity="denied" confidentiality="
       optional"/>
 </Outgoing>
 <Incoming value="authorized">
   <Attributes authentication="required" integrity="optional" confidentiality="
       optional"/>
 </Incoming>
</Communication>
```

Les balises `<Outgoing>` et `<Incoming>` correspondent respectivement à l'envoi et à la réception de communications d'une entité vers une autre. Elles indiquent la façon de transmettre des messages entre deux entités sécurisées impliquées dans l'interaction mais indiquent aussi grâce à l'attribut `value` de la balise `<Outgoing>` et `<Incoming>` si ce type de communication est autorisé ou non.

Il existe trois *attributs de sécurité* qui permettent de spécifier les propriétés de sécurité des communications : Authentification (**A**), Intégrité (**I**), Confidentialité (**C**). Chacun de ces trois attributs de sécurité peut être spécifié dans l'un des trois modes suivants : Requis, Optionnel,

Interaction	Description
Création d'un runtime	Autorise ou non la création d'un runtime (machine virtuelle java) donné. `<RuntimeCreation value="authorized|denied"/>`
Création de nœud	Autorise ou non la création locale ou distante d'un nouveau nœud à l'intérieur d'un runtime. `<NodeCreation value="authorized|denied"/>`
Téléchargement du bytecode des classes	Autorise ou non le chargement de bytecode depuis une entité vers une autre entité. `<CodeLoading value="authorized|denied"/>`
Création d'un objet actif	Autorise ou non la création d'une nouvelle activité (objet actif) à l'intérieur d'un nœud local ou distant. `<ActiveObjectCreation value="authorized|denied"/>`
Migration	Autorise ou non la migration d'un objet actif existant vers une entité pouvant être locale ou distante `<ActiveObjectMigration value="authorized|denied"/>`
Appel de méthode	Autorise ou non la propagation d'un appel de méthode depuis l'entité courante vers l'ensemble des entités contenues dans la balise <To>. `<Request value="authorized|denied" />`
Réception de résultat	Autorise ou non la réception d'un résultat d'un appel de méthode. `<Reply value="authorized|denied"/>`
Surveillance	Autorise ou non la surveillance (observation) du contenu d'une entité. La fonction de surveillance n'a pas la même définition selon le type réel de l'entité surveillée. Pour les domaines/-runtimes/nœuds, cette fonction permet d'obtenir la liste des runtimes/nœuds/objets actifs. Pour les objets actifs, la fonction permet la surveillance de son activité (état de la queue des requêtes). `<Listing value="authorized|denied"/>`
Terminaison d'un runtime	Autorise ou non la terminaison d'un runtime `<RuntimeTerminate value="authorized|denied"/>`
Terminaison d'un objet actif	Autorise la terminaison d'un objet actif `<ActiveObjectTerminate value="authorized|denied"/>`

TAB. 5.2 – Liste des interactions

Interdit. Ils seront appliqués à toutes les interactions décrites précédemment. Par exemple, la partie de la règle exposée ci-dessus spécifie que concernant toutes les interactions qui peuvent engendrer une communication sortante (requête, réponse, migration), les transferts doivent être authentifiés, l'intégrité des données ne doit pas être assurée, mais cependant le chiffrement de ces données est optionnel. Il est également possible de spécifier ces attributs sur toutes les communications entrantes. Cette gestion asymétrique des attributs de sécurité permet de différencier les deux sortes de flux et ainsi d'augmenter encore l'adaptabilité du mécanisme de sécurité.

Chaque règle peut être interprétée différemment selon la position de l'entité courante dans la partie `identification` de cette dernière. En effet, selon sa position, l'entité sera considérée comme effectuant l'interaction ou comme la subissant. C'est un des rôles du gestionnaire de sécurité de rechercher dans une règle si l'entité qu'il représente est présente dans un des deux ensembles d'entités (`<From>` ou `<To>`) et selon sa position, d'interpréter la règle par rapport à l'interaction courante. Par exemple, concernant une entité voulant émettre un appel de méthode, le gestionnaire de sécurité extraira les attributs de sécurité de la balise `<Outgoing>`, si cette même entité se retrouve à recevoir un appel de méthode, cette fois, le gestionnaire de sécurité extraira les attributs de sécurité de la balise `<Incoming>`.

5.4.3 Conclusion

Les politiques de sécurité présentées dans cette section nous permettent de répondre aux besoins induits par la structure décentralisée de notre approche. Chaque règle est construite de manière autonome de sorte que le gestionnaire de sécurité puisse prendre des décisions locales sans avoir besoin de se référer à un serveur de politique de sécurité.

L'introduction au sein de la règle des propriétés {"Requis","Optionnel","Interdit"} permet d'accorder au gestionnaire de sécurité la possibilité d'adapter, au besoin, la règle en fonction des autres règles tout en tenant compte des critères exprimés dans cette dernière.

5.5 Négociation Dynamique d'une Politique

Notre infrastructure de sécurité est prévue pour fonctionner de façon décentralisée, sans aucune gestion centralisée qui assurerait la correction de toutes les politiques de sécurité contenues dans toutes les entités du système. Compte tenu de cette décentralisation et même pour une application simple, il est possible d'obtenir plusieurs politiques de sécurité actives sur plusieurs niveaux de sécurité pour une interaction donnée. Afin de prendre en compte leurs critères de sécurité, elles doivent être *combinées*, *vérifiées* et *négociées* dynamiquement.

Nous présentons, dans un premier temps, le protocole que nous avons développé pour l'établissement de sessions entre les entités sécurisées. Nous détaillerons ensuite les parties liées à la recherche, au filtrage et à la composition des politiques de sécurité afin d'obtenir une politique de sécurité pour une interaction donnée.

5.5.1 Protocole d'établissement de session

Le protocole décrit dans cette partie permet l'établissement d'une connexion sécurisée entre deux entités. Il s'agit d'une version basée sur le protocole *Transport Layer Security* (TLS) [31] normalisé par l'Internet Engineering Task Force. La phase de négociation des paramètres de sécurité a été modifiée afin d'être compatible avec les pré-requis de notre architecture de sécurité. La figure 5.10 schématise les échanges de messages conduisant à l'établissement d'une session.

Lors d'une interaction entre une entité cliente (celle qui initie l'interaction) vers une entité serveur (celle qui reçoit l'interaction) les phases suivantes sont mises en œuvre pour aboutir à une interaction sécurisée :

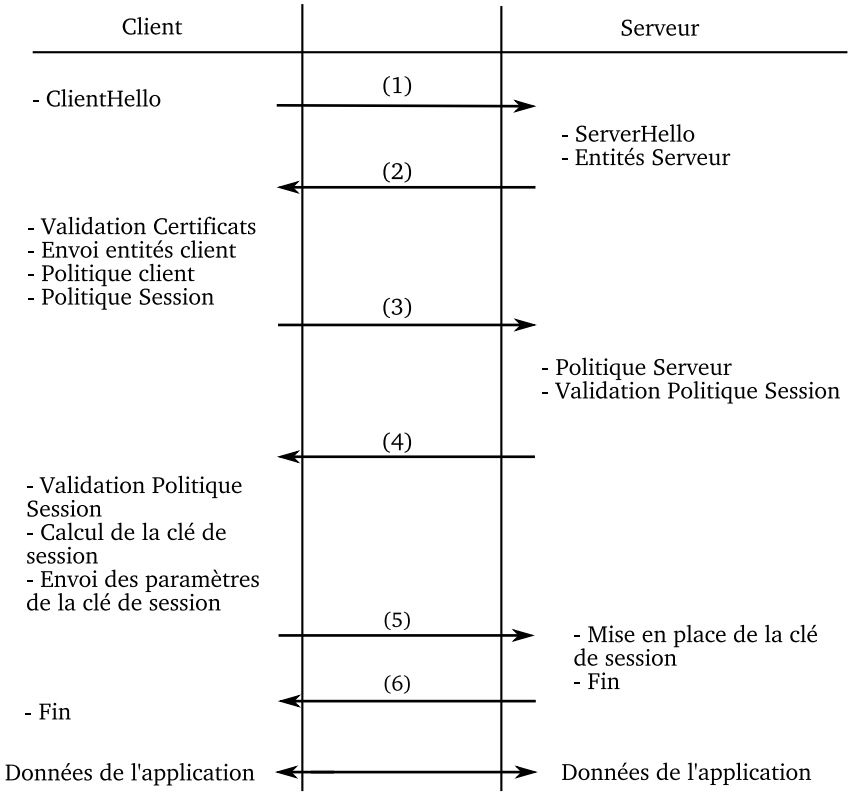


FIG. 5.10 – Protocole d'établissement de session

1. L'entité cliente débute une interaction sur l'entité serveur au travers de sa référence locale (le proxy).

   (a) Le proxy intercepte l'appel de méthode. Le code est toujours exécuté par le fil d'exécution de l'entité cliente. Le proxy permet d'accéder au code méta de l'entité cliente. C'est à partir de ce moment, que le mécanisme de sécurité peut débuter la phase de négociation d'une session. Il contacte l'entité serveur ce qui conduit à l'échange de message suivant :

   – [ClientHello] (1) : Ce message est envoyé quand une entité cliente se connecte pour la première fois à une entité serveur. Ce message comporte un identifiant de session (sessionID) qui permet d'identifier une session entre un client et un serveur. Cet identifiant sera placé dans tous les messages échangés entre le client et le serveur. Un identifiant de session ne devient valide qu'à la fin de la négociation lors de l'échange des messages [Fin]. Ce message contient aussi une variable *random* contenant une structure remplie aléatoirement par l'entité cliente.

   – [ServerHello] (2) : Ce message est envoyé en réponse au [ClientHello]. Il permet au serveur de créer une session représentant la connexion avec le client et ayant comme identifiant l'identifiant de session envoyé par le client. Ce message contient une variable *random* contenant une structure remplie aléatoirement par l'entité serveur.

   – [Entités Serveur] (2) : Ce message est composé d'une structure contenant les chaînes de certificats des entités englobant l'entité serveur (entités de niveau supérieur) ainsi que la chaîne de certificats de cette dernière.

   (b) Le mécanisme de sécurité de l'entité cliente également collecte toutes les informations

sur l'entité cliente (celles de l'entité elle-même et des entités englobantes). Une fois les informations de localisation de l'entité cliente et de l'entité serveur récupérées, le mécanisme de sécurité de l'entité cliente peut interroger toutes les entités englobantes afin d'obtenir les règles de sécurité à appliquer sur l'interaction.

(c) Le mécanisme de sécurité du client calcule la politique de sécurité résultante à partir de l'ensemble de politiques récupérées côté client.

2. Un message (3) est ensuite expédié à l'entité serveur contenant les chaînes de certificats identifiant les entités contenant l'entité cliente, la chaîne de certificats identifiant l'entité cliente et la politique de sécurité calculée par l'entité cliente.
3. L'entité serveur reçoit la politique de sécurité demandée par l'entité cliente. L'entité serveur :
   (a) récupère à son tour toutes les politiques de sécurité qui correspondent à l'interaction qui va avoir lieu, les combine afin d'obtenir une politique de sécurité résultante. Cette politique représente la politique de sécurité que requiert l'entité serveur.
   (b) compare la politique de sécurité requise par l'entité cliente à la politique calculée localement. Si ces politiques divergent, l'interaction est arrêtée et une exception est levée. Si les politiques sont identiques, un objet Session représentant l'interaction est créé. Si les politiques sont compatibles, elles doivent une dernière fois être combinées pour former la politique de sécurité finale avant de créer l'objet Session. Cet objet sera utilisé ultérieurement pour toutes les actions de sécurité (chiffrement, signature) qui pourront être requises pour cette interaction spécifique.
   (c) retourne à l'entité cliente (4) les informations de sécurité nécessaires (SessionID, politique de sécurité finale)
4. Le client vérifie que la politique retournée est bien compatible avec sa politique locale et crée, à son tour, un objet de type Session. Cet objet va exécuter les actions de sécurité requises par la politique de sécurité commune. L'entité cliente engendre une clé de session et l'envoie au serveur (5).
5. L'entité serveur finalise sa session en y ajoutant la clé de session et envoie le message [Fin] (6) à l'entité cliente.
6. l'entité cliente valide la session, et l'interaction peut être poursuivie.

Ce protocole d'établissement de session présuppose l'existence d'un mécanisme externe capable de récupérer et de sélectionner toutes les règles de sécurité sur une entité afin de les combiner entre elles dans le but de générer une politique de sécurité résultante intégrant les critères de toutes les politiques s'appliquant à l'interaction.

En effet, il est possible pour un acteur donné d'écrire une politique de sécurité générale pour son application et de vouloir une politique de sécurité précise pour un sous-ensemble de sa politique de sécurité générale. L'algorithme de recherche le plus simple aurait été d'arrêter la recherche à la première politique rencontrée qui correspond à l'interaction. Cette solution simple et peu coûteuse en ressources système possède plusieurs inconvénients. Tout d'abord, le fait qu'une seule règle puisse être sélectionnée impose au concepteur de la politique de sécurité de décrire tous les cas possibles qu'il voudrait prendre en compte et d'ordonner les politiques de sécurité selon un ordre précis. Deuxièmement, la dynamicité de l'approche laisse supposer que des cas non prévus initialement par le concepteur de la politique puissent mener à l'utilisation des règles trop génériques. Pour toutes ces raisons, nous avons choisi de doter notre mécanisme de sécurité d'algorithmes plus intelligents. Ces algorithmes sont au nombre de trois : l'algorithme de sélection des politiques de sécurité, l'algorithme de filtrage et en dernier l'algorithme de composition des politiques de sécurité afin d'obtenir une politique de sécurité finale pour une interaction donnée.

5.5.2 Algorithme de sélection

Lors d'une interaction, plusieurs entités peuvent être impliquées du fait de la structuration hiérarchique de notre modèle (figure 5.11).

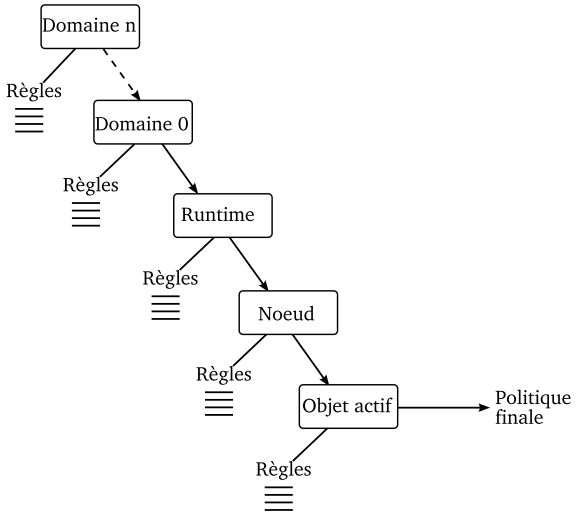


FIG. 5.11 – Niveaux de sécurité hiérarchiques

Chaque gestionnaire de sécurité de chaque entité doit rechercher la ou les politiques de sécurité les plus appropriées selon les parties en présence. Nous avons pour cela développé l'algorithme de sélection adapté aux particularités de nos politiques de sécurité.

L'algorithme de sélection a pour fonction la recherche et la sélection des politiques de sécurité correspondantes aux entités impliquées dans une interaction donnée. La recherche se fait sur les champs correspondant aux balises `<From>` et `<To>` du fichier de politiques de sécurité associé à l'entité courante.

Une règle est considérée comme valide lorsqu'à la fois les ensembles sources et cibles correspondent aux entités actuellement impliquées dans l'interaction. Une entité est considérée comme valide si son certificat ou un des certificats composant sa chaîne de certificats correspondent à une entité dans l'ensemble des entités recherchées.

5.5.3 Algorithme de filtrage

L'algorithme de filtrage doit comme son nom l'indique appliquer un filtre sur toutes les politiques de sécurité sélectionnée par l'algorithme de sélection. En appliquant le filtre seulement sur les politiques déjà sélectionnées, on réduit l'ensemble de politiques à comparer et ainsi le temps d'établissement d'une session. Pour une entité donnée, l'algorithme de filtrage ne retient que les politiques de sécurité les plus adéquates parmi celles sélectionnées. Pour chaque politique de sécurité, il faut comparer les ensembles d'entités initiant l'interaction (entités sources) et les ensembles d'entités sur lesquels est effectuée l'interaction (entités cibles) et vérifier qu'une des autres règles sélectionnées n'est pas une règle plus spécifique que la règle courante.

Soit S_1 (resp. T_1) l'ensemble des entités sources (resp. entités cibles) de la politiques P_1 et S_2 (resp. T_2) l'ensemble des entités sources (resp. entités cibles) de la politique P_2. P_1 est dit plus spécifique que P_2 si l'un de ces points est validé :

$$S_1 \subset S_2 \wedge T_1 \subset T_2$$
$$S_2 = S_1 \wedge T_2 \subset T_1$$
$$S_2 \subset S_1 \wedge T_2 = T_1$$

Il est possible de comparer deux entités seulement si elles appartiennent à la même chaîne de certification. Dans ce cas, l'entité la plus spécifique sera celle dont le certificat dérive du certificat de l'autre entité. Soit la chaîne de certification suivante `Alice -> Bob` où Alice représente

le certificat racine de la chaîne, s'il existe deux règles, la première contenant uniquement l'entité `Alice` dans sa partie `<To>` et la deuxième contenant `Bob` dans cette même partie, alors la deuxième règle est dite plus spécifique que la première et sera la seule conservée.

5.5.4 Algorithme de composition

Le dernier algorithme est celui de la composition des politiques de sécurité. Les deux algorithmes précédents ont permis de sélectionner un ensemble de politiques de sécurité correspondant à une interaction donnée. Il faut maintenant pouvoir combiner ces politiques de sécurité pour obtenir une seule politique qui exprimera les critères de sécurité de l'interaction. L'algorithme de composition se base sur les propriétés attribuées à chaque interaction décrite dans chaque politique de sécurité pour combiner les politiques entre elles. L'algorithme est exposé au sein de la figure 5.12.

		Entité A		
		Requis (+)	Optionel (?)	Interdit (-)
Entité B	Requis (+)	+	+	Echec
	Optionel (?)	+	?	-
	Interdit (-)	Echec	-	-

FIG. 5.12 – Algorithme de composition

Selon les diverses combinaisons possibles, l'interaction peut être :
- autorisée (cas + et cas ?). En effet, le cas optionnel indique que l'interaction se fera en fonction des propriétés des autres règles. Si les autres règles ne sont pas plus strictes, le mécanisme de sécurité n'activera pas cette fonctionnalité.
- interdite (cas -). Dans ce cas, l'interaction est bloquée par le mécanisme de sécurité.
- en échec. Ce cas survient lorsque deux règles expriment des propriétés contraires sur une interaction donnée. Dans un tel cas, l'interaction n'est pas autorisée.

Cet algorithme de composition assure l'interopération des politiques de sécurité initiales (voir section 2.6). Si une politique de sécurité résultante est trouvée alors elle préserve les attributs de sécurité des politiques initiales.

5.6 Sécurité induite par le déploiement

Déployer une application sur des grilles de calcul consiste à déployer un ensemble de processus collaboratifs (activités) sur un grand nombre d'ordinateurs. La politique de sécurité qui doit être appliquée sur ces processus dépend de plusieurs facteurs et notamment du type de déploiement utilisé. Il apparaît évident que le déploiement d'une application au sein d'un seul site (réseau local) d'une grille de calcul n'implique pas les mêmes risques au niveau de la sécurité que le déploiement de cette même application sur divers sites de cette grille surtout s'ils sont interconnectés au moyen d'un réseau public comme Internet.

De plus, lors de la conception d'un programme, le programmeur ne peut pas savoir à l'avance comment et sur quelle infrastructure son programme va être déployé. Sa seule possibilité est de prévoir une structuration de son programme suivant une architecture virtuelle qui correspondra le plus possible aux schémas de déploiement de son application.

Ainsi, la politique de sécurité d'une application ne devrait pas être fortement liée au code de l'application mais plutôt exprimée en dehors du code source d'une application, de sorte qu'elle soit facilement configurable par l'utilisateur.

5.6.1 Principe

Nous avons déjà présenté les descripteurs de déploiement (voir section 4.6) qui introduisent un niveau d'abstraction concernant le déploiement de l'application et permettent une structuration de l'application en terme d'activités conceptuelles. Nous allons étendre cette approche en se servant de la structuration de l'application pour exprimer les besoins de sécurité de chaque activité et ainsi définir une politique de sécurité générale pour l'application.

Afin de définir de manière abstraite la politique de sécurité d'une application, nous réutilisons la notion de nœuds virtuels introduite précédemment et nous posons la définition suivante :

Définition 12 *Sécurité des nœuds virtuels*
Une politique de sécurité peut être définie au niveau des nœuds virtuels. A l'exécution, la politique de sécurité d'un nœud va dépendre de la politique du nœud virtuel dont il fait partie et de la politique de la JVM sur laquelle il se trouve.

Les nœuds virtuels sont le support de l'expression de la politique de sécurité d'une application. Si, au moment de la conception d'une application, un processus requiert un niveau de sécurité spécifique (authentification des utilisateurs et chiffrement des communications), alors ce processus doit être rattaché à un nœud virtuel sur lequel ces règles de sécurité sont imposées. Il est de la responsabilité du concepteur de structurer son application ou ses composants en utilisant l'abstraction liée aux nœuds virtuels pour imposer les règles de sécurité voulues.
Lorsqu'une politique de sécurité doit être calculée pour une interaction donnée, le nœud, en tant qu'entité sécurisée, va rechercher dans ses propres politiques de sécurité celles qui correspondent à l'interaction. Etant donné, que le nœud appartient à un nœud virtuel donné d'une application donnée et que sa politique de sécurité est identique à celle de l'application, la politique de sécurité qu'il retournera sera conforme à celle de l'application. Nous avons également vu que l'algorithme de composition des politiques de sécurité assure leur interopérabilité. Ainsi, nous pouvons prouver que quel que soit le résultat du déploiement d'une application, la politique de sécurité initialement définie au niveau des nœuds virtuels sera conservée et appliquée.

La figure 5.13 expose les divers éléments entrant en jeu dans le processus de déploiement d'une application sécurisée. Elle présente une application structurée autour de deux nœuds virtuels (`VN1` et `VN2`), chacun contenant un objet actif. Le fichier de déploiement spécifie deux runtimes (`Runtime1` et `Runtime2`). Le premier `Runtime1` est déployé avec la politique de sécurité de l'application et apparaîtra comme une entité appartenant à l'application. Le deuxième `Runtime2` est un runtime déjà existant, possédant sa propre politique de sécurité, indépendante de celle de l'application. Les nœuds et les objets actifs dépendront de la politique de sécurité de l'application décrite dans un fichier externe référencé par le descripteur de déploiement. Le référencement se fait par l'ajout d'une balise supplémentaire au sein du descripteur de déploiement :

```
<security file="politiquesDeLApplication.xml"/>
```

5.6.2 Contexte et environnement

Lors du déploiement d'une application, l'utilisateur va être amené à écrire des politiques de sécurité concernant les interactions entre les entités de l'application elles mêmes et les interactions entre les entités de l'application et des entités extérieures au contexte de l'application. Pour écrire une règle d'une politique de sécurité, il faut pouvoir identifier les entités ; nous avons vu qu'il était possible d'utiliser le certificat de l'entité pour cette identification. Cependant, les diverses entités d'une application, ne possèdent pas, dans le cas d'un déploiement sécurisé transparent, de certificats. Ces derniers ne seront créés dynamiquement que lors du déploiement de l'application.

La seule identification des diverses entités qui existe à ce niveau du déploiement de l'application est celle donnée par le descripteur de déploiement. On retrouve dans le descripteur des identifiants permettant la mise en place de l'architecture qui servira de support à l'application. Ce sont ces identifiants que nous allons réutiliser au sein du fichier de politique de sécurité.

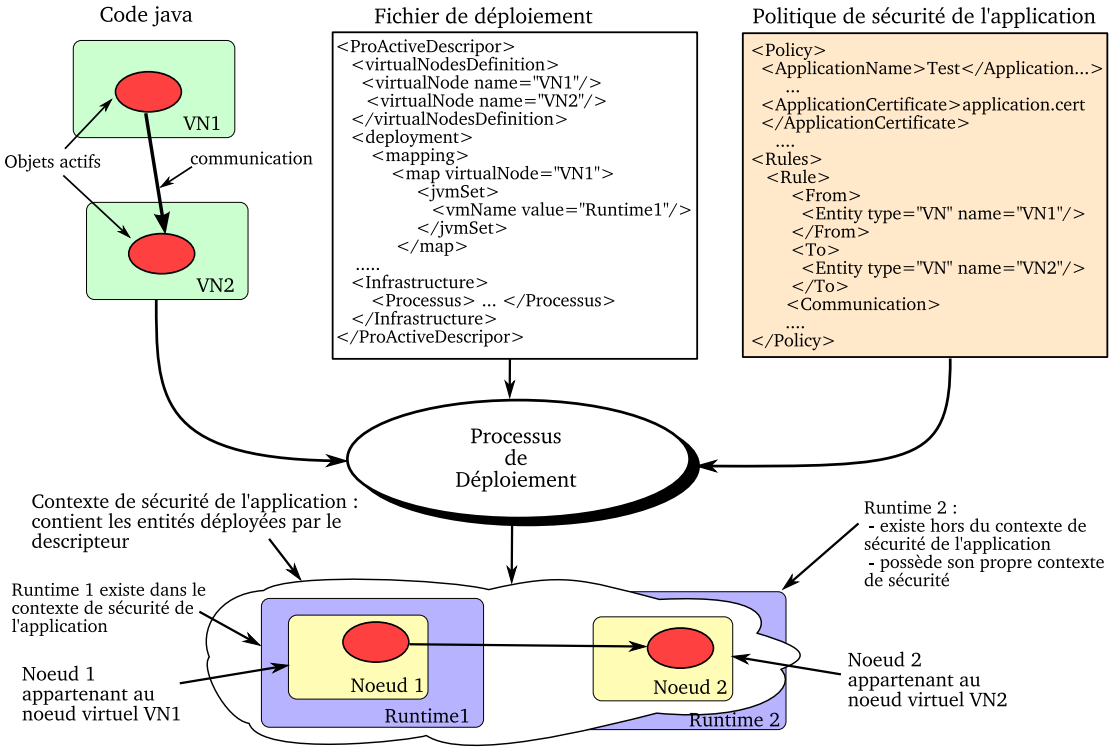


FIG. 5.13 – Déploiement d'une application sécurisée

5.6.3 Descripteur de politiques de sécurité

Nous allons décrire maintenant la syntaxe du descripteur de politiques de sécurité. Il s'agit du fichier qui contient les informations nécessaires au mécanisme de sécurité pour mettre en place l'architecture sécurisée et la politique de sécurité qui va gouverner le comportement de l'application. Le fichier est au format XML.

Un fichier de politique de sécurité commence par la balise XML `<Policy>`. La première partie du fichier contient les informations utiles au mécanisme de sécurité.

La première balise permet de nommer l'application, ce nom se trouvera au sein de toutes les entités créées à partir de ce déploiement. Il permet à l'utilisateur, s'il le souhaite, de nommer une instance donnée d'une application juste en changeant la chaîne de caractères.

```
<ApplicationName>Garden</ApplicationName>
```

La partie suivante concerne l'identification de l'instance de l'application associée à ce descripteur de sécurité, deux approches sont possibles. La première consiste à référencer le certificat de l'utilisateur en utilisant la ligne XML suivante :

```
<UserCertificate>user-certificate.p12</UserCertificate>
```

où `user-certificate.p12` représente le fichier contenant le certificat de l'utilisateur.

Cette approche a l'avantage de minimiser les actions imposées à l'utilisateur lors du lancement de son application. En effet, le certificat d'application sera généré automatiquement à partir de celui de l'utilisateur. L'inconvénient vient justement de cette transparence, le certificat étant généré automatiquement, il sera impossible à l'utilisateur de conserver ce certificat s'il le désire.

Pour palier à cette situation, il est possible de spécifier le certificat à utiliser comme certificat d'application. C'est-à-dire le certificat qui servira de certificat racine pour toutes les entités créées par le mécanisme de sécurité. Dans ce cas, on remplace la ligne XML précédente par :

```
<Certificate>application-certificate.p12</Certificate>
```

La validation d'un certificat demande de faire confiance à l'autorité qui a généré le certificat. Par défaut, seuls les certificats contenus dans la chaîne de certification qui a permis de générer le certificat de l'application sont considérés comme entité de confiance. Il est possible de rajouter des certificats jouant le rôle d'autorités de certification en utilisant le code XML suivant :

```
<TrustedCertificationAuthority>
  <CertificationAuthority>
    <Certificate>autorite1.cert</Certificate>
  </CertificationAuthority>
  <CertificationAuthority>
    <Certificate>autorite2.cert</Certificate>
  </CertificationAuthority>
</TrustedCertificationAuthority>
```

où `autorite1.cert` et `autorite2.cert` désignent les fichiers contenant les certificats des autorités de certification.

En *ProActive* la thread `main` du code exécuté pendant le déploiement (au lancement de l'application) ne dépend pas d'un objet actif. Afin de lui permettre d'effectuer des appels asynchrones vers un objet actif un `HalfBody` est créé implicitement. Du point de vue de la sécurité, ce `HalfBody` doit être une entité sécurisée s'il veut pouvoir effectuer des interactions, notamment la création de nœuds ou d'objets actifs. Dans le but de rendre transparente l'utilisation du `HalfBody`, le comportement par défaut génère automatiquement une entité sécurisée et l'associe à ce `HalfBody`. Cette entité sécurisée possède tous les droits envers les autres entités qui possèdent le même certificat d'application. Il est cependant possible d'affecter un nom représentant un nœud virtuel à cette entité afin de pouvoir exprimer des règles de sécurité la concernant. L'affectation se fait en définissant la balise suivante :

```
<MainVirtualNode>VN_Main</MainVirtualNode>
```

La suite du descripteur de sécurité contient les règles de sécurité et utilise la syntaxe que nous avons déjà présentée à la section 5.4. L'exemple ci-après montre comment imposer des communications chiffrée entre les deux objets actifs lors d'une communication initiée à partir des objets localisés sur le nœud virtuel `VN1` et adressée aux objets localisés sur le nœud virtuel `VN2`.

```
<Rules>
  <Rule>
      <From>
         <Entity type="VirtualNode" name="VN1"/>
      </From>
      <To>
         <Entity type="VirtualNode" name="VN2"/>
      </To>
      <Communication>
      <Outgoing value="authorized">
         <Attributes authentication="required" integrity="required"
             confidentiality="required"/>
      </Outgoing>
      <Incoming value="authorized">
         <Attributes authentication="required" integrity="required"
             confidentiality="required"/>
      </Incoming>
      </Communication>
      <Migration value="authorized"/>
      <AOCreation value="authorized"/>
      <Request value="authorized"/>
      <Reply value="authorized"/>
   </Rule>
</Rules>
```

Le code ci-après présente une implantation possible de l'exemple. On retrouve la création des deux objets actifs sur les deux nœuds `VN1` et `VN2`, ainsi qu'une communication entre ces deux objets.

```
// creation et activation du descripteur
ProActiveDescriptor pad = ProActive.getProactiveDescriptor("file:"+args[0]);
pad.activateMappings();

// creation du premier objet actif sur le noeud virtuel VN1
Flower a = (Flower) ProActive.newActive(Flower.class.getName(), new Object[]{"
    Amaryllis"},proActiveDescriptor.getVirtualNode("VN1").getNode());

// creation du deuxieme objet actif sur le noeud virtuel VN2
Flower b = (Flower) ProActive.newActive(Flower.class.getName(), new Object[]{"
    Amaryllis"},proActiveDescriptor.getVirtualNode("VN1").getNode());

// etablissement d'une communication
a.receiveReference(b);
```

La communication entre les deux objets `a` et `b` se fera selon la politique de sécurité décrite dans le fichier de configuration associé soit une communication authentifiée, chiffrée et intègre.

5.6.4 C3D : application de rendu collaboratif

C3D est une application de rendu collaboratif. Elle permet à plusieurs utilisateurs de visualiser la même scène en 3 dimensions. Chaque utilisateur peut agir sur la scène en rajoutant des objets ou en modifiant l'angle de la caméra. Le rendu de la scène est effectué par un ou plusieurs moteurs de lancer de rayons.

L'architecture de C3D (figure 5.14) se compose de trois types d'objets actifs :

- *L'interface graphique* qui permet d'afficher le rendu de la scène et aux utilisateurs d'agir sur cette même scène.
- Le *contrôleur* qui coordonne les actions effectuées par les utilisateurs. Il possède des références vers les moteurs de rendu qui serviront à calculer la scène 3D. Le contrôleur découpe la scène en autant de morceaux qu'il existe de moteurs disponibles. Le contrôleur est contenu dans un nœud virtuel nommé `VN_controleur`.
- Le *moteur de rendu* dont le rôle est de calculer une partie de la scène 3D. Tous les moteurs sont contenus dans un seul nœud virtuel nommé `VN_moteur`.

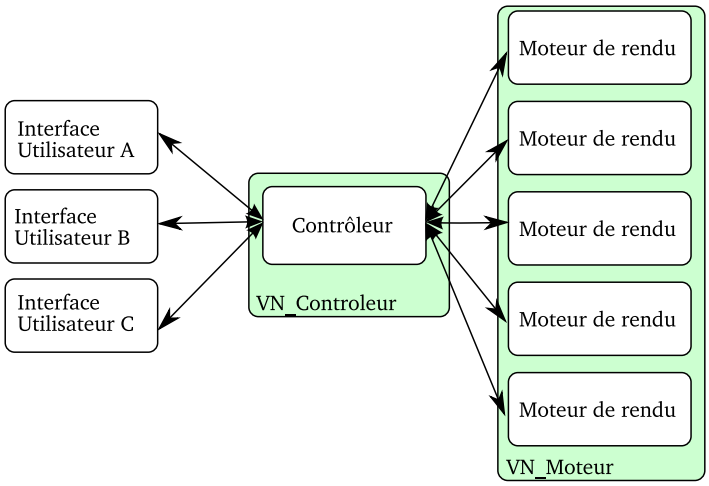


FIG. 5.14 – Architecture de C3D

Supposons qu'un utilisateur `A` veuille déployer les moteurs de rendu sur plusieurs grappes de calcul, dont une est locale a son entreprise et ne nécessite pas de sécurité particulière, par contre la deuxième se trouve dans un laboratoire externe, la connexion se faisant par un réseau public non sécurisé (figure 5.15). Parallèlement, l'utilisateur `B` est autorisé à visualiser la scène 3D. `B` possède un ordinateur au sein de l'organisation `O`. Lorsqu'il accède à la scène 3D depuis cet ordinateur seule l'authentification est nécessaire.

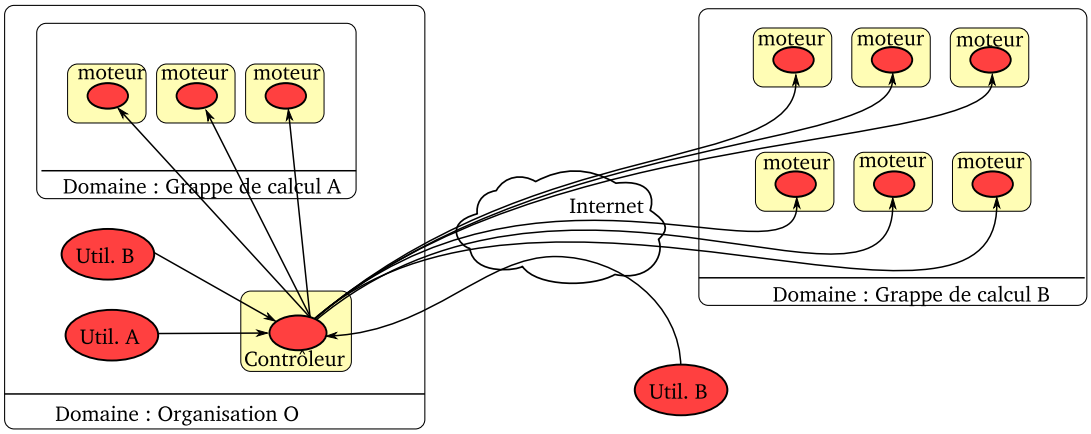


FIG. 5.15 – C3D déployée sur plusieurs grappes

Les politiques de sécurité sont les suivantes :

1. Le contrôleur (nœud virtuel `VN_controleur`) peut accéder aux moteurs de rendu `VN_moteur`.

```
<From>
 <Entity type="VirtualNode" name="VN_Controleur"/>
</From>
<To>
 <Entity type="VirtualNode" name="VN_Moteur"/>
</To>
<Request value="authorized"/>
<Reply value="authorized"/>
```

2. Les moteurs de rendu sont autorisés à migrer au sein de leur nœud virtuel.

```
<From>
 <Entity type="VirtualNode" name="VN_Moteur"/>
</From>
<To>
 <Entity type="VirtualNode" name="VN_Moteur"/>
</To>
<ActiveObjectMigration value="authorized"/>
```

3. Toute interaction entre le domaine de l'organisation `O` et celui de la grappe de calcul `B` doit se faire de manière authentifiée, intègre et confidentielle.

```
<From>
 <Entity type="file" name="organisationO.pkcs12"/>
</From>
<To>
 <Entity type="file" name="grilleDeCalculB.pkcs12"/>
</To>
<Communication>
  <Outgoing value="authorized">
    <Attributes authentication="required" integrity="optional" confidentiality
        ="optional"/>
  </Outgoing>
  <Incoming value="authorized">
    <Attributes authentication="required" integrity="optional" confidentiality
        ="optional"/>
  </Incoming>
</Communication>
Request value="authorized"/>
<Reply value="authorized"/>
```

4. L'utilisateur A est autorisé à se connecter au contrôleur.

```
<From>
 <Entity type="file" name="UtilisateurA.pkcs12"/>
</From>
<To>
 <Entity type="VirtualNode" name="VN_Controleur"/>
</To>
<Communication>
  <Outgoing value="authorized">
    <Attributes authentication="optional" integrity="optional" confidentiality
        ="optional"/>
  </Outgoing>
  <Incoming value="authorized">
    <Attributes authentication="optional" integrity="optional" confidentiality
        ="optional"/>
  </Incoming>
</Communication>
<Request value="authorized"/>
<Reply value="authorized"/>
```

5. L'utilisateur B est autorisé à se connecter :
   (a) depuis l'intérieur de l'organisation O, l'authentification suffit.

```
<From>
 <Entity type="file" name="UtilisateurB.pkcs12"/>
 <Entity type="file" name="organisationO.pkcs12"/>
</From>
<To>
 <Entity type="VirtualNode" name="VN_Controleur"/>
</To>
<Communication>
  <Outgoing value="authorized">
    <Attributes authentication="required" integrity="optional"
        confidentiality="optional"/>
  </Outgoing>
  <Incoming value="authorized">
    <Attributes authentication="required" integrity="optional"
        confidentiality="optional"/>
  </Incoming>
</Communication>
<Request value="authorized"/>
<Reply value="authorized"/>
```

   (b) sinon la communication doit être authentifiée, intègre et confidentielle.

```
<From>
 <Entity type="file" name="UtilisateurB.pkcs12"/>
</From>
<To>
 <Entity type="VirtualNode" name="VN_Controleur"/>
</To>
<Communication>
  <Outgoing value="authorized">
    <Attributes authentication="required" integrity="required"
        confidentiality="required"/>
  </Outgoing>
  <Incoming value="authorized">
    <Attributes authentication="required" integrity="required"
        confidentiality="required"/>
  </Incoming>
</Communication>
```

```
<Request value="authorized"/>
<Reply value="authorized"/>
```

Suivant ces politiques de sécurité, lorsqu'une scène va être rendue les données sont normalement transférées vers les moteurs de rendu par le contrôleur. Cependant selon leur localisation (grappe de calcul A ou B) les attributs de sécurité différeront. Les communications vers les moteurs de rendu de la grappe de calcul A se feront selon la règle 1 tandis que les communications vers les moteurs de rendu de la grille de calcul B se feront suivant la composition des règles 1 et 3.

La migration d'un moteur de rendu au sein d'un ordinateur vers un autre de la même grappe se fera selon la règle 2 tandis que la migration de ce même moteur de rendu entre les deux grappes se fera selon la composition des règles 2 et 3. Si un moteur de rendu de la grappe de calcul B migre vers la grappe de calcul A, les futures communications entre ce moteur de rendu et le contrôleur se feront selon la règle 1. Parallèlement, si un moteur de rendu de la grappe de calcul A migre vers la grappe de calcul B, les futures communications entre ce moteur de rendu et le contrôleur se feront selon la composition des règles 1 et 3.

Si l'utilisateur B se trouve au sein de la grappe de calcul B, les communications se feront selon la composition des règles 3 et 5b.

5.6.5 Conclusion

L'externalisation du mécanisme de définition des politiques de sécurité permet d'adapter la politique de sécurité d'une application au type de déploiement utilisé. Cette approche facilite le déploiement d'une infrastructure sécurisée et la mise en place d'un contexte de sécurité autour d'une application qui n'a pas conscience d'être sécurisée.

5.7 Propagation dynamique du contexte de sécurité

La section précédente a permis de présenter l'intégration de la sécurité au sein du processus de déploiement. L'approche permet la mise en place d'une architecture sécurisée lors du déploiement d'une application afin d'établir un contexte de sécurité autour de l'application. Cependant une application est une entité dynamique et, lors de son exécution, elle peut être amenée à créer de nouveaux runtimes, nœuds, objets actifs. Le mécanisme présenté dans cette section s'inscrit dans la continuité du mécanisme de déploiement d'une infrastructure sécurisée par le biais des descripteurs de déploiement. Ce mécanisme permet d'assurer le confinement des nouveaux éléments de l'application et la propagation de la politique de sécurité à ces nouveaux éléments.

On s'intéresse essentiellement à une application dont la politique de sécurité est imposée lors de son déploiement. Cependant, le mécanisme que nous allons décrire est aussi compatible avec toute application qui gère explicitement sa sécurité en utilisant notre mécanisme de sécurité.

Tout code applicatif est exécuté sous l'identité d'une entité sécurisée (un objet actif). Une entité sécurisée est conçue pour être autonome et pouvoir gérer de manière décentralisée sa propre sécurité. Elle doit aussi s'assurer que le code applicatif ne va pas introduire des failles dans la sécurité de l'application, notamment lors de la création de nouveaux objets. Parmi les objets que le code applicatif est capable de créer, on retrouve :

– les objets Java, qu'ils soient actifs ou passifs ;
– les nœuds qui sont un élément de l'application.

Un objet passif ne possède pas de gestionnaire de sécurité, il dépend du gestionnaire de sécurité de l'entité sécurisée qui le contient et qui lui impose sa politique de sécurité. La création d'un objet passif ne requiert aucune interaction de la part du mécanisme de sécurité.

À l'inverse, un objet actif ou un nœud sont des entités sécurisées qui possèdent un gestionnaire de sécurité et une identité (un certificat). Lorsque du code applicatif instancie un objet actif, le mécanisme de sécurité doit pouvoir, à partir des informations locales, générer une nouvelle entité sécurisée qui devra apparaître comme appartenant à la même application.

Chaque entité sécurisée possède non seulement son propre certificat, mais aussi le certificat et la clé privée de l'application à laquelle elle appartient. À partir de ce certificat d'application, il est possible de générer un certificat pour l'entité qui va être créée. En accord avec notre modèle d'authentification, deux entités sécurisées appartiennent à la même application si et seulement si les deux certificats dérivent du même certificat d'application. Ainsi la nouvelle entité sécurisée créée appartiendra au même contexte de sécurité que l'entité sécurisée à partir de laquelle elle a été créée.

Par exemple, la création d'un objet actif est réalisée au moyen de l'instruction suivante :

```
A a = (A) ProActive.newActive("A", {...});
```

Cette instruction déclenche le processus de création de l'objet actif sur le nœud. Ce processus inclut la création des méta objets du body et notamment le gestionnaire de sécurité. Les étapes effectuées par le mécanisme de sécurité sont présentées au sein de la figure 5.16, le pseudo-code ci-après présente la logique du mécanisme :

```
si ((Entité Sécurisée) && (création autorisée) alors
  - Créer un certificat à partir du certificat d'application de
    l'entité sécurisé courante
  - Dupliquer les règles de sécurité
  - Générer un gestionnaire de sécurité
finsi
```

Un mécanisme similaire est mis en place pour la création d'un nœud.

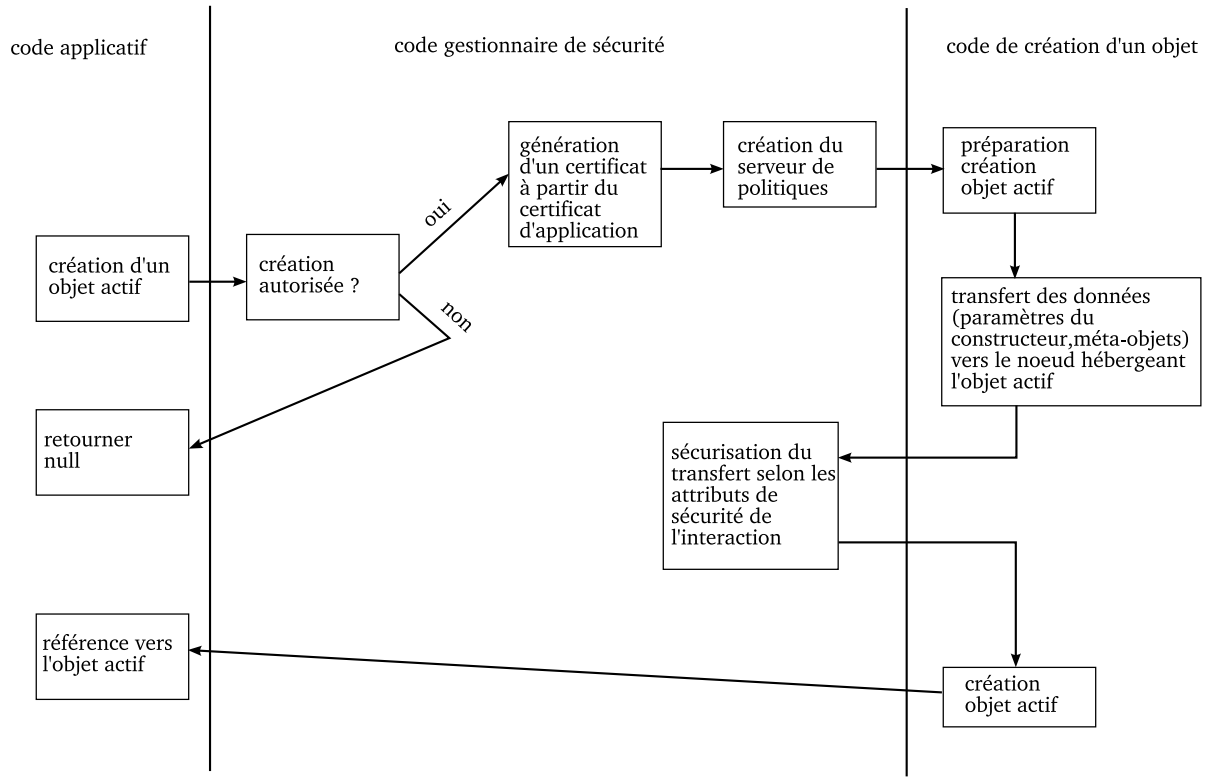


FIG. 5.16 – Création d'un objet actif

Ainsi lors de la création d'une entité sécurisée, les politiques de sécurité de l'entité qui la crée sont répliquées sur cette dernière.

5.8 Exemple du journal intime

Cette section reprend l'application "Journal Intime" présentée lors de l'état de l'art et présente son implantation en utilisant notre modèle de sécurité. L'application se décompose en deux parties, le serveur et le client. Le serveur est un objet actif, positionné sur un nœud appartenant au nœud virtuel `VN_Serveur`. Jusqu'à présent, nous avons supposé que le déploiement de l'application était orchestré d'un seul endroit, dans la méthode principale (`main`) du programme. Dans cet exemple, nous avons choisi de déployer dans un premier temps le serveur avec sa politique de sécurité propre, puis le client lui aussi avec une politique de sécurité propre. De plus, le client ne sera pas explicitement créé comme un objet actif : sa sécurité reposera sur les mécanismes de sécurité du `HalfBody` qui entrent en jeu lors d'une telle situation.

La classe `DiaryImpl` représente l'implantation du code du serveur. La méthode exposée `addEntry` est celle qui permet de d'ajouter des messages dans le journal.

```
public class DiaryImpl implements Diary, Serializable {
/*... */
public void addEntry(String entry) {
  entries.add(entry);
}
```

La méthode `main` présente le code de lancement du serveur. Ce dernier est positionné sur un nœud dépendant du nœud virtuel `VN_Serveur`, le serveur est enregistré sur l'adresse "`//localhostMyDiary`".

```
public static void main(String[] args) {
   Diary diary = null;
   try {
     // activation du descripteur de deploiement
     ProActiveDescriptor pad = ProActive.getProactiveDescriptor(args[0]);
     pad.activateMappings();

     // creation de l'objet actif jouant le role du serveur
     diary = (Diary) ProActive.newActive("org.objectweb.proactive.examples.mydiary
         .DiaryImpl", new Object[] { }, pad.getVirtualNode("VN_Serveur").getNode()
         );

     // enregistrement de l'URL du serveur
     ProActive.register(diary, "//localhost/MyDiary");
   } catch (Exception e) {
     e.printStackTrace();
   }
}
```

Le descripteur de sécurité associé au descripteur de déploiement spécifie la politique de sécurité à appliquer aux échanges sécurisés entre le nœud virtuel contenant le serveur (`VN_Serveur`) et celui contenant le client (`VN_Client`). Le fait que le ou les nœuds appartenant au nœud virtuel `VN_Client` ne soient pas connues lors du déploiement ne pose aucun problème de configuration. Ces nœuds seront créés ultérieurement par la partie cliente de l'application qui devra, bien évidemment, posséder le même certificat d'application que celui qui a été utilisé pour la partie serveur de l'application.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy>
 <ApplicationName>JournalIntime</ApplicationName>
 <Certificate>ji.pkcs12</Certificate>
 <Rules>
   <Rule>
       <From>
         <Entity type="VirtualNode" name="VN_Client"/>
       </From>
       <To>
         <Entity type="VirtualNode" name="VN_Serveur"/>
```

```
      </To>
      <Communication>
      <Outgoing value="authorized">
         <Attributes authentication="required" integrity="required"
             confidentiality="required"/>
      </Outgoing>
      <Incoming value="authorized">
         <Attributes authentication="required" integrity="required"
             confidentiality="required"/>
      </Incoming>
      </Communication>
      <Request value="authorized"/>
      <Reply value="authorized"/>
   </Rule>
 </Rules>
</Policy>
```

Le code du client ne comporte qu'une thread `main` ; le `HalfBody` associé gérera les interactions. Nous pouvons noter que le descripteur de déploiement ne sert ici qu'à positionner l'identité de ce code, il n'y a pas de déploiement à proprement parler.

```
public static void main(String[] args) {
  try {
    // activation du descripteur de deploiement
    ProActiveDescriptor pad = ProActive.getProactiveDescriptor(args[0]);
    pad.activateMappings();

    // recuperation de la reference vers le serveur
    Diary remoteDiary = (Diary) ProActive.lookupActive("org.objectweb.proactive.
        examples.mydiary.Diary", "//localhost/MyDiray" );

    // execution des methodes
    remoteDiary.addEntry("une nouvelle entree dans mon journal");
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Le certificat d'application utilisé est le même que celui utilisé pour le serveur. Le certificat représentant la thread `main` sera issu du même certificat d'application que celui du serveur. Les deux entités (le `main` et l'objet serveur) seront donc vues comme appartenant à la même application.

```
<Policy>
 <ApplicationName>JournalIntime</ApplicationName>
 <Certificate>ji.pkcs12</Certificate>
 <MainVirtualNode>VN_Client</MainVirtualNode>
 <Rules>
   <Rule>
      <From>
        <Entity type="DefaultVirtualNode" name=""/>
      </From>
      <To>
        <Entity type="DefaultVirtualNode" name=""/>
      </To>
      <Communication>
      <Outgoing value="authorized">
         <Attributes authentication="required" integrity="required"
             confidentiality="required"/>
      </Outgoing>
      <Incoming value="authorized">
         <Attributes authentication="required" integrity="required"
             confidentiality="required"/>
```

```
      </Incoming>
      </Communication>
      <Request value="authorized"/>
      <Reply value="authorized"/>
   </Rule>
 </Rules>
</Policy>
```

5.9 Conclusion

Nous avons, dans ce chapitre, présenté un modèle de sécurité ainsi qu'une architecture de sécurité qui permettent de résoudre les deux principaux problèmes que nous avons identifiés lors de notre étude bibliographique qui sont la gestion de la dynamicité des applications et la transparence du mécanisme de sécurité vis-à-vis du code métier de l'application. Le modèle présuppose que les hôtes sur lesquels le code de la bibliothèque sera exécuté sont de confiance.

La première étape a consisté à concevoir un mécanisme de sécurité hiérarchique en se basant sur le principe du moniteur de référence qui requiert deux notions essentielles :
- l'identification des sujets et des objets,
- des politiques de sécurité exprimant ce que d'authentification permettant d'identifier les diverses entités existantes afin de les intégrer à l'architecture de sécurité.

En ce qui concerne l'identification, nous nous sommes basés sur une architecture à clé publique de type SPKI afin d'attribuer à chaque entité une identité propre. La particularité du modèle réside dans l'utilisation d'un certificat d'application définissant un contexte d'application. Ce contexte permet de différencier plusieurs instances de la même application.

Du fait de la nature hiérarchique et décentralisée de notre modèle, plusieurs politiques de sécurité peuvent s'appliquer sur une interaction donnée. Nous avons donc défini un langage de politiques de sécurité adaptable, autorisant la composition de plusieurs politiques de sécurité pour former une politique résultante qui sera appliquée à une interaction donnée.

La configuration de la politique de sécurité de l'application se fait de manière déclarative dans des fichiers de configuration sans demander de modification du code de l'application. Ces fichiers sont lus lors du déploiement de l'application. Il est ainsi possible d'adapter la politique de sécurité de l'application en fonction du type de déploiement choisi (une seule machine, une grappe de calcul, grille de calcul interconnectée par Internet, ...).

Hormis la gestion des besoins de sécurité fondamentaux (contrôle d'accès, chiffrement et intégrité des communications), le mécanisme de sécurité est également capable de s'adapter dynamiquement à des nouvelles situations survenant dans le cycle de vie d'une application distribuée, notamment l'acquisition de nouvelles ressources et la création de nouveaux objets.

Le mécanisme de propagation dynamique du contexte de sécurité que nous avons introduit permet d'étendre automatiquement le contexte de sécurité initial présent lors du lancement d'une application à tous les objets qui seront créés lors de son exécution.

Chapitre 6

Implantation dans ProActive : une approche transparente

Le code applicatif se base sur les fonctionnalités de *ProActive* et de son interface de programmation pour se déployer, communiquer et gérer la migration de ses activités. Les différentes parties de *ProActive* s'organisent en couche distinctes (fig 6.1). Chaque couche fournit des services et utilise ceux de la couche supérieure. La partie *code applicatif* (couche 4) représente le code de l'application et son comportement propre. Il s'agit du code écrit par le développeur d'une application. La couche 3 représente la partie qui gère les différents comportements fournis par *ProActive* à l'application (migration, tolérance aux pannes, communications de groupe). On voit sur le schéma que cette couche chevauche un morceau du code applicatif. Ce chevauchement représente les interfaces de programmation offertes par la librairie pour permettre au code applicatif d'accéder à ses fonctionnalités. Intercalée entre la couche des fonctionnalités de la librairie et celle des mécanismes de transport se trouve la couche qui va gérer la sécurité de la bibliothèque. Les pointillés précise que cette couche est transparente vis-à-vis de sa couche supérieure et de sa couche inférieure, la couche 3 a l'impression d'utiliser directement la couche 1 et inversement. Cette solution nous permet de limiter les interactions entre le mécanisme de sécurité et les autres mécanismes de la librairie.

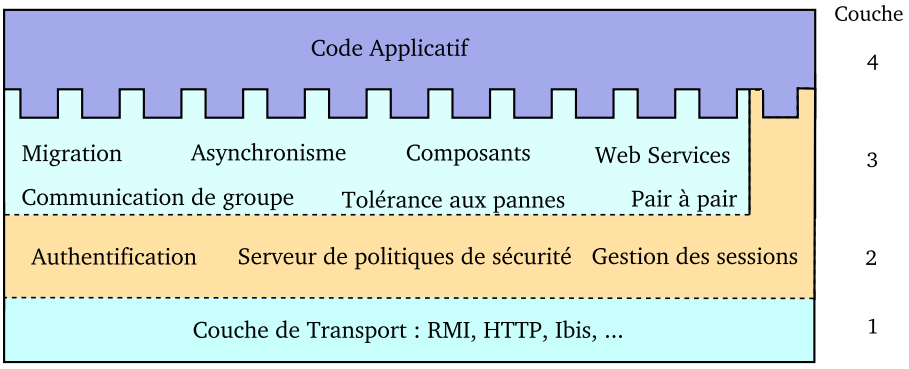


FIG. 6.1 – Diagramme des couches de protocoles de ProActive

Dans ce chapitre, nous allons présenter dans un premier temps l'implantation du modèle de sécurité au sein des divers composants de la bibliothèque, puis la façon dont le mécanisme de sécurité s'intègre avec les autres fonctionnalités et mécanismes de la bibliothèque.

6.1 Création des entités sécurisées

6.1.1 Le runtime sécurisé

Comme toute entité sécurisée, un runtime possède une identité et une politique de sécurité. Ces paramètres peuvent être imposés au runtime soit lors de sa création (cas 1 et 2) comme paramètres du programme qui lance le runtime, soit lors de la création effective du runtime, dans le code (cas 3).

1. en utilisant les descripteurs de déploiement. Il est possible lors de la définition du processus qui lancera un runtime de spécifier le descripteur de sécurité que le runtime devra utiliser.

   ```
   <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess">
     <jvmParameters>
      <parameter value="-Dproactive.runtime.security=runtime_policy.xml" />
     </jvmParameters>
   </jvmProcess>
   ```

2. en utilisant les scripts de démarrage fournis avec la bibliothèque. Par exemple, sous Linux la commande serait la suivante :

   ```
   # ./startRuntime.sh runtime_policy.xml
   ```

3. en utilisant de manière fonctionnelle l'interface de programmation prévue à cet effet.

   ```
   // creation du gestionnaire de securite
   // le descripteur de securite "runtime_policy.xml" contient les parametres
   // du runtime securise
   ProActiveSecurityManager psm = new ProActiveSecurityManager("runtime_policy.
       xml");

   // creation du runtime et affectation du gestionnaire de securite
   ((ProActiveRuntimeImpl)ProActiveRuntimeImpl.getDefaultRuntime()).
       setSecurityManager(psm);
   ```

   le fichier "runtime_policy.xml" contient le descripteur de sécurité du runtime, Cette commande démarre un runtime et l'enregistre sur une URL donnée qui peut être définie statiquement ou créer dynamiquement le processus. Par la suite, le runtime être référencé dans un descripteur de déploiement via son URL.

Cependant, contrairement à un nœud dont la création est due à un appel de méthode sur un runtime, la création d'un runtime implique la création d'un nouveau processus qui peut être local ou distant via l'utilisation d'une commande externe à la bibliothèque et à la plate-forme Java. Nous sommes confrontés au problème de transfert des paramètres de sécurité (identité, politiques de sécurité) entre les processus. La solution actuelle suppose que les fichiers requis aient été préalablement copiés dans des endroits accessibles par chaque runtime. Ces fichiers ne sont nécessaire que lors du démarrage du runtime. Un fois, le runtime démarré ces fichiers peuvent être supprimés.

6.1.2 Le nœud sécurisé

La sémantique de création d'un nœud distant a dû être revue et modifiée pour coller aux contraintes de sécurité. Initialement, lorsqu'un objet demandait la création d'un nombre x de nœuds à un runtime, le runtime créait les nœuds et renvoyait les URLs des nœuds à l'objet ayant initié la demande. On se retrouvait avec des nœuds sur lesquels il était impossible de spécifier une politique de sécurité propre. La politique de sécurité de ces nœuds ne pouvaient dépendre que du runtime qui les accueillait.

Après modification, la sémantique sécurisée permet d'introduire le fait que, lors de l'appel de méthode demandant la création du nœud distant, un paramètre supplémentaire, le gestionnaire de sécurité associé à ce nœud, puisse être passé.

```
// creation d'une regle de securite
PolicyRule policyRule = new PolicyRule();

// creation d'un serveur de politique et ajout des regles
PolicyRule[] policyRules = { policyRule };
PolicyServer policyServer = new PolicyServer(policyRules);

// chargement du certificat de l'objet
// le mot de passe du keystore est : "albator"
KeyStore keyStore = KeyStore.getInstance("PKCS12", "BC");
keyStore.load(new FileInputStream("application.p12"),"albator".toCharArray());

// creation du gestionnaire de securite
ProActiveSecurityManager psm =
        new ProActiveSecurityManager(keyStore,policyServer);

// creation du noeud sur le runtime
runtime.createLocalNode("nodeName",false, psm,"vnName", "jobId");
```

Si ce paramètre n'est pas spécifié, le mécanisme de sécurité récupérera la politique de sécurité de l'entité sécurisée appelante et l'imposera comme politique de sécurité du futur nœud. Il en sera de même pour l'identité du futur nœud qui dépendra du certificat d'application de l'entité appelante.

La sécurisation de l'appel de méthode `createLocalNode` et le transfert sécurisé des paramètres sur le runtime distant sont assurés étant donné que le code de création du nœud dépend d'une entité sécurisée. Ainsi une session sera établie entre l'entité sécurisée appelante et le runtime distant, et l'appel de méthode se fera selon la politique de sécurité négociée.

6.1.3 L'objet actif sécurisé

Au sein d'un objet actif, la sécurité est implantée de manière orthogonale et transparente au code source de l'application. Elle est définie tel un nouveau comportement au sein de l'objet actif. Elle s'insère auprès du body d'un objet actif sous la forme d'un nouveau méta objet, le gestionnaire de sécurité.

Nous avons jusqu'à présent abordé la sécurité comme étant transparente au code de l'application. Il est cependant possible d'instancier un objet sécurisé sans déléguer au mécanisme de sécurité le soin de créer les objets de sécurité nécessaires.

La première étape consiste à créer le serveur de politique de sécurité. Sa construction requiert un tableau de politique de sécurité. Dans l'exemple présenté ici, le serveur ne contiendra qu'une seule règle. Cette règle sera une règle générique, comme l'autorise le mécanisme de sécurité. Les champs `<From>` et `<To>` désignant les entités source et cible sont remplis avec une entité par défaut qui, lors de la comparaison avec une autre entité, répond toujours vrai. Les champs correspondant aux attributs de communication sont positionnés à leur valeur optionnelle. Tous les autres champs de la règle sont remplis avec les valeurs autorisant les interactions.

```
// creation d'une regle de securite
PolicyRule policyRule = new PolicyRule();

// creation d'un serveur de politique et ajout des regles
PolicyRule[] policyRules = { policyRule };
PolicyServer policyServer = new PolicyServer(policyRules);
```

La deuxième étape consiste à charger le certificat de l'entité. Nous supposerons ici que le certificat a déjà été généré.

```
// chargement du certificat de l'objet
// le mot de passe du keystore est : "albator"
KeyStore keyStore = KeyStore.getInstance("PKCS12", "BC");
keyStore.load(new FileInputStream("application.p12"), "albator".toCharArray());
```

La troisième étape consiste à créer le gestionnaire de sécurité à partir des objets créés précédemment et de l'insérer au sein de l'ensemble de méta-objets qui seront passés à l'objet actif lors de sa création.

```
// creation du gestionnaire de securite
ProActiveSecurityManager psm =
        new ProActiveSecurityManager(keyStore,policyServer);

// creation des meta objets du futur objet actif
MetaObjectFactory factory = ProActiveMetaObjectFactory.newInstance();

// ajout du gestionnaire de securite aux meta objets
factory.setProActiveSecurityManager(psm);
```

La dernière étape permet la création de l'objet actif sur le nœud par défaut en lui passant l'ensemble des méta objets nécessaires à sa création.

```
// pas de parametres, appel du constructeur par defaut de l'objet
Object[] params = { } ;

// creation de l'objet sur le noeud par default
Node node = NodeFactory.getDefaultNode();

// creation de l'objet actif
A a = (A) ProActive.newActive("A", params, node , null, factory);
```

Cette approche permet la création d'un objet actif sécurisé à partir d'un environnement où le mécanisme de sécurité n'a pas été initialisé. Il est intéressant de noter que ce nouvel objet actif étant une entité sécurisé, le mécanisme de propagation dynamique du contexte de sécurité vu en 5.7 peut entrer en jeu. Dans l'hypothèse où une méthode de l'objet actif A lance la création d'un nouvel objet actif b de type B sans notion explicite de sécurité :

```
public class A {
  ...
  public Object createActiveObject() {
    ...
    B b = (B) ProActive.newActive("B", new Object[] {}, null);
    ...
  }
  ...
}
```

alors étant donné que l'objet A contient un gestionnaire de sécurité et dans le cas présent, un certificat d'application, le mécanisme de propagation dynamique intercepte le processus de création de l'objet actif B, génère un certificat d'entité à partir du certificat d'application et lui affecte les politiques de sécurité de l'objet A.

6.1.4 Les domaines de sécurité

Un domaine est, par définition, un serveur de politiques de sécurité qui permet de regrouper sous un même contexte de sécurité des entités sécurisées. En tant que serveur, il doit être accessible à distance par les entités qu'il supervise. Cependant, son accès doit être restreint aux seules entités qui en ont le droit. En effet, son interrogation par des tiers non autorisés pourrait leur permettre de déceler des failles dans la politique de sécurité du domaine.
En résumé un domaine doit être :

- accessible à distance ;
- sécurisé.

Il s'agit tout simplement de la définition d'un objet actif sécurisé. Les étapes pour la création d'un domaine sont les suivantes :

La première étape consiste à créer le gestionnaire de sécurité qui va servir à protéger l'objet actif représentant le domaine de sécurité.

```
// le fichier "domainPSM.xml" contient les informations
// necessaires a la creation du gestionnaire de securite
ProActiveSecurityManager psm = new ProActiveSecurityManager("domainPSM.xml");

// creation des meta objets du futur objet actif
MetaObjectFactory factory = ProActiveMetaObjectFactory.newInstance();

// ajout du gestionnaire de securite aux meta objets
factory.setProActiveSecurityManager(psm);
```

Ensuite vient la création de l'objet actif, sur le nœud par défaut, en lui passant l'ensemble des méta objets nécessaires à sa création. Le paramètre passé à l'objet actif correspond au fichier contenant les politiques de sécurité du domaine. Ce fichier peut être différent de celui qui a servi à spécifier les politiques de sécurité propres à l'objet actif. Il est ainsi possible d'écrire des politiques de sécurité spécifiques pour les entités sécurisées contenues dans le domaine et de spécifier d'autres politiques de sécurité pour l'entité sécurisée représentant le domaine dont les besoins en termes de sécurité peuvent être différents.

```
// creation de l'objet sur le noeud par default
Node node = NodeFactory.getDefaultNode();

// le fichier "Domaine_A.xml" contient les politiques de securite
// du domaine.
Object[] params = { "Domaine_A.xml" } ;


// creation de l'objet actif
Domain domainA = (Domain) ProActive.newActive("org.objectweb.proactive.ext.
    security.domain.Domain", params, node , null, factory);
```

6.2 Interface de programmation

Nous avons pour le moment présenté l'intégration transparente de la sécurité. Afin de compléter notre modèle, nous avons ajouté une interface de programmation (API) permettant à un utilisateur, s'il le désire, d'intégrer des primitives de sécurité à son code afin d'obtenir des informations relatives à la sécurité de son application.

Lors de l'exécution d'un objet actif sécurisé, le programmeur peut avoir besoin d'accéder à diverses informations concernant l'identité sous laquelle son code s'exécute.

```
// retourne le certificat de l'objet actif
ProActiveSecurity.getMyCertificate();

// retourne la chaine de certification de l'objet actif
ProActiveSecurity.getMyCertificateChain();

// retourne la cle prive de l'objet actif
ProActiveSecurity.getMyPrivateKey();
```

Selon la localisation de l'objet actif, le programme peut adopter un comportement différent. La méthode

```
// retourne la hierarchie des entites englobant l'objet
ProActiveSecurity.getEntities();
```

permet d'obtenir la hiérarchie des entités contenant l'objet actif.

Hormis les primitives concernant l'identité et la localisation, le programme peut avoir besoin de savoir quelle est l'entité qui a exécuté l'appel menant à l'exécution de la méthode. Pour cela, le programmeur dispose du jeu d'instructions suivant :

```
// retourne le certificat de l'entite qui a fait l'appel
// sur la methode courante
ProActiveSecurity.getCurrentMethodCallerCertificate();

// retourne la politique de securite de la session
// qui a ete negociee pour recevoir la methode courante
ProActiveSecurity.getCurrentMethodPolicy();

// retourne les entites transmises par l'appelant
ProActiveSecurity.getCallerEntities();
```

6.3 Migration et sécurité

La migration d'activité est une fonctionnalité présente dans de nombreux intergiciels. Elle permet de solutionner plusieurs limitations liées à la programmation distribuée. L'étude bibliographie nous a permis de présenter différentes solutions de sécurité existantes dans le cadre des agents mobiles. Pour le mécanisme de sécurité présenté dans cette thèse, nous sommes partis du postulat que nous nous trouvions dans une architecture avec des hôtes (runtimes) de confiance. Dans notre approche, un hôte sera considéré de confiance s'il est capable de s'authentifier.

6.3.1 Contexte de sécurité et migration

Le contexte de sécurité d'une application contient toutes les entités créées lors de son déploiement, ainsi que celles créées dynamiquement par l'application. Nous voulons exposer les implications de la localisation de l'objet actif avant sa migration et après sa migration sur le contexte de sécurité d'une application. Les cas que nous allons présenter sont regroupés au sein de la figure 6.2.

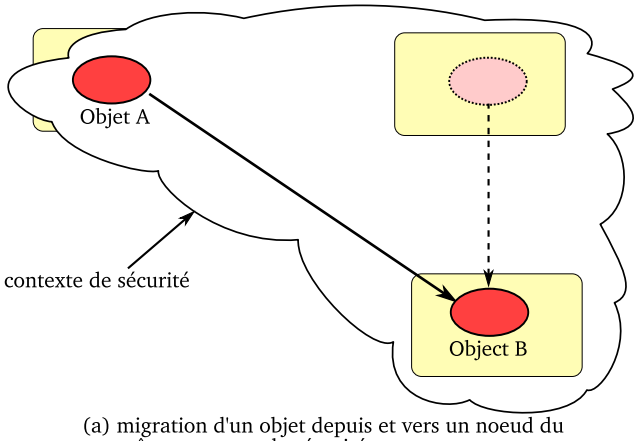
Le premier schéma (a) présente le cas le plus simple. Il s'agit de la migration d'un objet actif de et vers un nœud appartenant au même contexte de sécurité, c'est-à-dire à la même application. Cette configuration se retrouve généralement lors du déploiement d'une application au moyen d'un descripteur de déploiement.

Le deuxième schéma (b) présente la migration d'un objet actif depuis un nœud appartenant au même contexte vers un nœud qui n'appartient pas au même contexte. Dans cette configuration, le contexte de sécurité de l'application est présent à la fois sur le nœud contenant l'objet avant sa migration et sur l'objet actif. La migration de cet objet actif entraîne la propagation du contexte de sécurité vers la nouvelle localisation de l'objet actif.

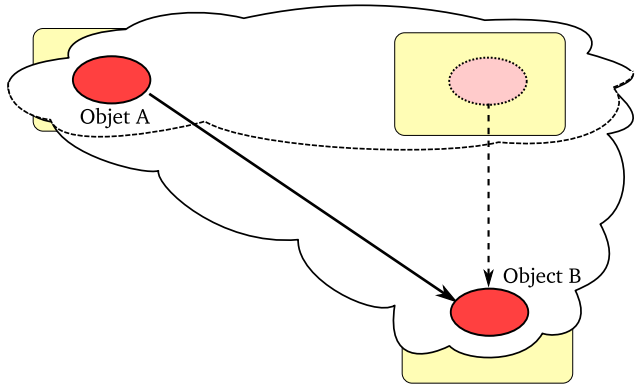
Le dernier schéma (c) présente la migration d'un objet actif depuis et vers un nœud dont les contextes de sécurité ne sont pas les mêmes que ceux de l'application dont dépend l'objet. La migration de cet objet actif entraîne le déplacement de la partie du contexte de sécurité de l'application contenu dans l'objet actif de l'ancienne localisation vers la nouvelle.

6.3.2 Migration sécurisée

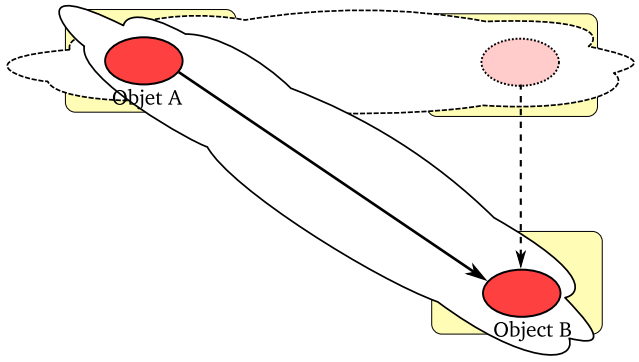
La politique de sécurité à appliquer pour une interaction entre deux objets actifs peut avoir changé après une migration. Les politiques de sécurité des entités sécurisées (nœud, runtime, domaines) qui vont contenir l'objet actif peuvent être différentes de celles localisées à l'ancien emplacement de l'objet actif. Par conséquent, les politiques de sécurité associées aux sessions déjà existantes peuvent être fausses. La migration d'un objet actif impose de ré-établir les sessions en cours, donc de renégocier les politiques de sécurité.



(a) migration d'un objet depuis et vers un noeud du même contexte de sécurité



(b) migration depuis un noeud du même contexte de sécurité vers un noeud externe



(c) migration de et vers des noeuds n'appartenant pas au même contexte de sécurité que l'objet

FIG. 6.2 – Divers cas de migration d'une application sécurisée

Les entités qui possèdent des sessions établies avec l'objet qui a migré ne sont pas informées de la migration de ce dernier. On aurait pu choisir d'informer toutes les entités tierces de la migration d'un objet et ainsi leur faire invalider leur session. Cette approche possède plusieurs inconvénients :

1. il faut garder des références vers toutes les entités qui ont communiqué avec l'entité qui a migré, ce qui provoque une augmentation de l'occupation mémoire du gestionnaire de sécurité ;
2. les communications engendrées pour diffuser l'information qu'une migration a eu lieu prennent du temps et des ressources systèmes (occupation de la bande passante du réseau).

L'approche que nous avons choisie se positionne comme une approche plus paresseuse (*lazy*). Plutôt que d'informer les entités qui ont une session établie avec l'objet mobile, ces entités découvriront, lors leur prochaine interaction avec l'entité mobile que cette dernière a migré. Le respect des politiques de sécurité est assuré par le fait que s'il existait une session la communication qui doit révéler que l'objet a migré se fera selon la politique de la session. Après la mise à jour de la localisation de l'entité cible, une nouvelle session va pouvoir être négociée et la communication pourra être transmise en accord avec les nouveaux paramètres de sécurité.

Lors de la migration d'un objet actif, deux mécanismes de localisation peuvent être utilisés, le répéteur ou le serveur de localisation (voir section 4.4).

L'utilisation du serveur de localisation (figure 6.3) est la méthode la plus facile à gérer au niveau de la sécurité. Lorsqu'un objet a migré, un appel de méthode sur celui-ci lève une exception qui sera attrapée au niveau du protocole de communication et qui déclenchera la recherche de la nouvelle localisation via le serveur de localisation. Au niveau du protocole de sécurité, l'échec de l'appel de méthode et la recherche via le serveur de localisation vont avoir pour conséquence l'invalidation de la session. Il est intéressant de noter que le serveur de localisation étant un objet actif toutes les communications vers et au départ de celui-ci peuvent être sécurisées.

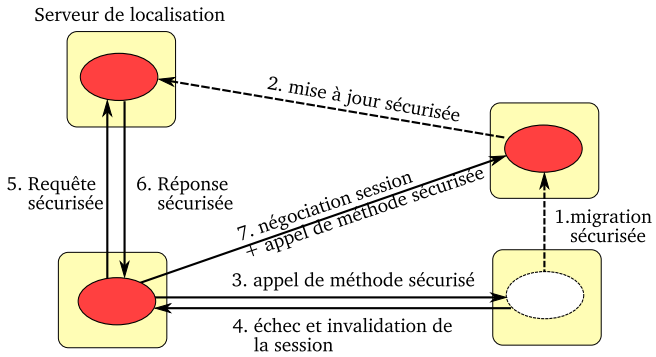


FIG. 6.3 – Serveur de localisation et sécurité

L'utilisation des répéteurs (figure 6.4) et des mécanismes de sécurité est plus délicate. Il n'y a pas, comme dans le mécanisme avec serveur de localisation, de détection de la migration. Dans le cas sans sécurité, le répéteur se contente de transmettre la requête à l'objet actif. Dans le cas avec sécurité, la transmission de la requête de l'émetteur au répéteur se fait de manière normale en respectant la politique de sécurité négociée. Cependant, une fois sur le répéteur, le message sécurisé ne peut pas être acheminé directement vers l'objet mobile. En effet, une telle action pourrait entraîner une brèche au niveau du protocole de sécurité si la politique de sécurité utilisée pour envoyer le message vers l'ancienne localisation de l'objet diffère de celle qui aurait du être utilisée pour envoyer le message de l'objet émetteur vers le nouvel emplacement de l'objet qui a migré. Pour éviter cette brèche, le répéteur a été modifié afin de pouvoir avertir l'objet appelant de la migration de l'objet appelé. Une fois averti, l'objet appelant peut établir une nouvelle session vers l'objet appelé qui reflétera les nouveaux paramètres de sécurité. En présence d'une chaîne

de répéteur, le comportement est identique, le premier répéteur informera l'objet appelant de la migration de l'objet appelé. Le premier message du protocole d'établissement de session suivra la chaîne de répéteurs, la réponse a ce premier message d'établissement de session permettra de mettre à jour la référence que possède l'objet appelant sur l'objet appelé.

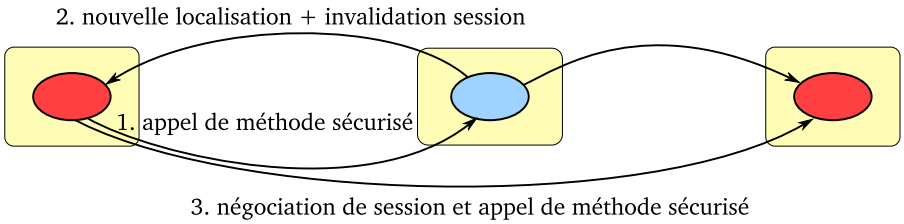


FIG. 6.4 – Répéteurs et sécurité

Dans les deux cas, l'entité appelante est informée de la migration lors de la phase de *rendez-vous* existant au début de chaque communication. Le fait d'informer l'appelant lors de la phase de rendez-vous permet de mettre à jour ses données tout en restant dans sa phase d'envoi de message et ainsi de renvoyer le message sécurisé de manière transparente et sans effet de bord sur les autres parties de l'entité.

6.3.3 Optimisation au sein des grilles de calcul

Parmi les architectures matérielles utilisables par la bibliothèque, nous avons les grilles de calcul. Cette configuration matérielle possède des spécificités propres. Du point de vue de la sécurité, les ordinateurs d'une grille de calcul sont généralement soumis aux mêmes contraintes de sécurité, aux mêmes politiques de sécurité. Techniquement, l'utilisation d'un domaine de sécurité permet de remplir aisément ces contraintes.

D'autre part, le déploiement d'une application sur une grille de calcul ne s'oppose en rien au fait que tous les nœuds soient vus de manière équivalente en terme de sécurité. Le déploiement d'un nœud virtuel sur les machines de la grille va imposer la même politique de sécurité à tous les nœuds de la grille.

Par défaut, la migration d'un objet actif a pour conséquence l'invalidation de ses politiques de sécurité. Cependant, il se peut que la migration d'un objet actif, si les deux conditions précédemment énoncées se révèlent exactes, se fasse au sein des mêmes contextes de sécurité. Dans un tel cas, les sessions précédemment négociées resteraient valides. La phase d'invalidation des sessions et de renégociation serait surperflue. Il est intéressant d'optimiser la migration d'un objet actif au sein des mêmes contextes de sécurité. Pour cela, le mécanisme de migration a été étendu pour prendre en compte les deux suivants.

– Si l'objet migre depuis un ensemble d'entité `A` et dont le nœud appartient au nœud virtuel `VN` vers le même ensemble d'entité `A` et dont le nœud cible appartient aussi au nœud virtuel `VN` alors l'objet conserve ses sessions préalablement établies.
– Dans tout autre cas, les sessions sont invalidées et après migration, toutes les sessions devront être renégociées.

6.4 Communications de groupe et sécurité

Il existe trois sortes de groupes : simples, hiérarchiques et actifs. Il existe un polymorphisme entre tous ces groupes ce qui permet d'utiliser les mêmes interfaces de programmation pour pro-

grammer à l'aide des groupes.

Une communication de groupe correspond à une communication *ProActive* standard vers les n membres du groupe avec une première optimisation au moment de la construction de l'objet `MethodCall` pour éviter n fois la sérialisation des paramètres de l'appel de méthode ainsi qu'une deuxième optimisation étant donné que les appels sont exécutés en parallèle.

Il existe cependant une différence dans la sémantique des communications sécurisées lorsqu'un membre du groupe est un objet actif (groupe simple), soit un autre groupe (groupe hiérarchique), soit un objet actif représentant un autre groupe (*groupe actif*). Nous allons maintenant détailler ces trois cas.

6.4.1 Groupe simple

Un groupe simple est un groupe contenant des objets actifs ou des objets passifs. Lorsqu'un appel de méthode vers un tel groupe est propagé à chaque membre du groupe, le mécanisme de sécurité va intercepter l'appel, comme lors d'un appel de méthode *ProActive* standard, et initier le calcul de la politique de sécurité à appliquer pour l'appel de méthode vers le membre du groupe qui doit recevoir l'appel de méthode. Du point de vue de la sécurité, une communication vers un groupe simple se résume à une suite d'appels de méthode vers chaque membre du groupe. Ce processus est résumé au sein de la figure 6.5.

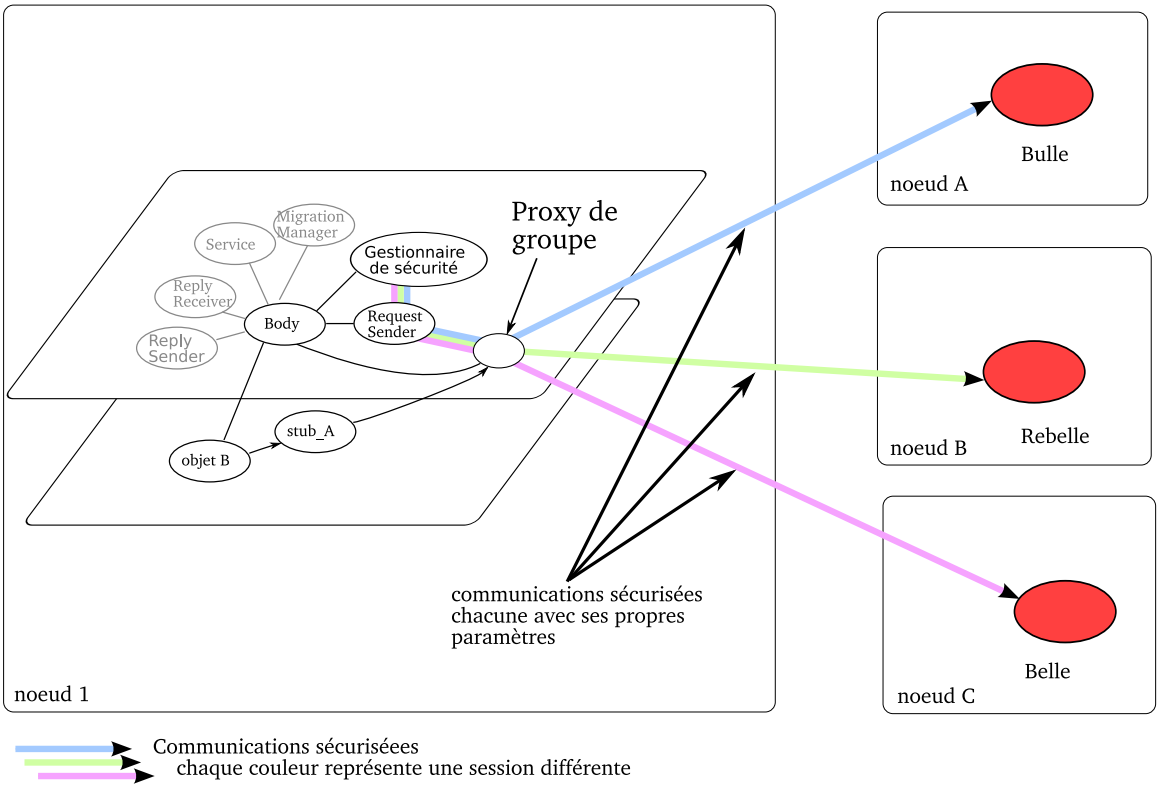


FIG. 6.5 – Groupe simple et sécurité

6.4.2 Groupe hiérarchique

Un groupe hiérarchique est un groupe contenant d'autres groupes. Cette approche facilite la structuration d'applications distribuées. Son interaction et son intégration avec le mécanisme de sécurité se fait de manière transparente. Lors d'un appel de méthode sur un groupe hiérarchique,

le comportement est identique à une communication de groupe faite sur un groupe simple, le mécanisme de sécurité ne fait pas de distinction entre les objets d'un groupe qu'il soit hiérarchique ou pas. Sur l'exemple de la figure 6.6, l'objet actif appelant établit une session sécurisée avec chaque objet actif du groupe1 qui comporte deux sous groupes (a et b).

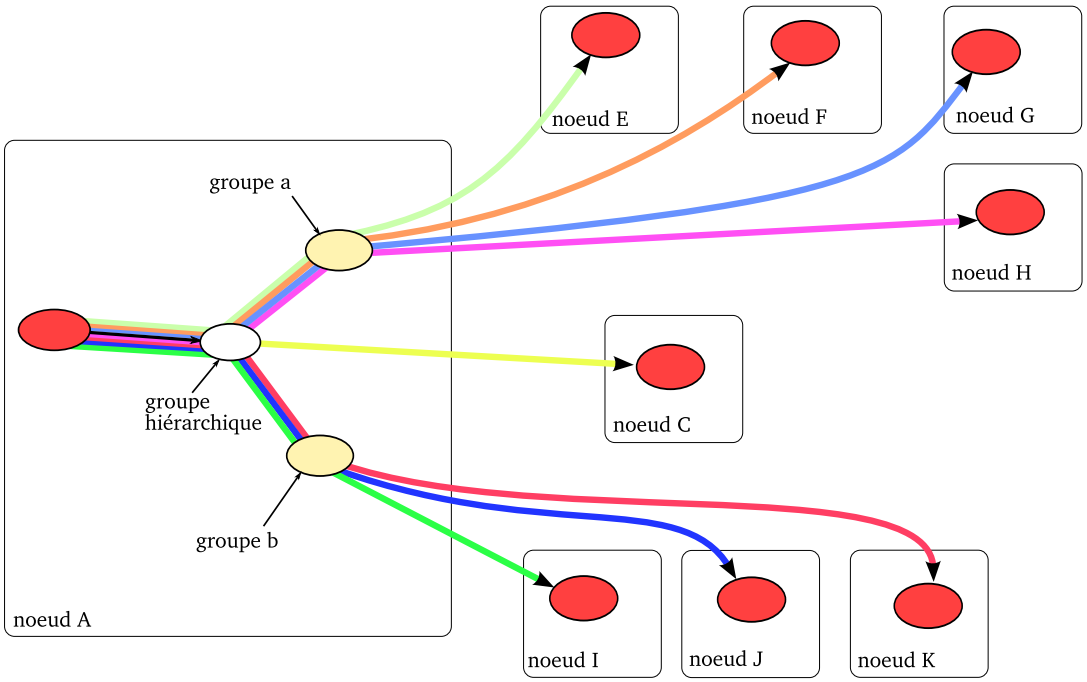


FIG. 6.6 – Groupe hiérarchique et sécurité

6.4.3 Groupe actif

La technique employée pour rendre un groupe accessible à distance est de le rendre actif, en utilisant les primitives fournies par l'interface de programmation de la bibliothèque. Dans ce cas, le groupe étant un objet actif, il possède tous les méta objets que possède un objet actif et notamment un gestionnaire de sécurité. De ce fait, le groupe actif est vu par le gestionnaire de sécurité comme un objet actif à part entière.

Lors d'un appel de méthode vers un tel groupe (figure 6.7), le gestionnaire de sécurité de l'objet actif appelant va établir une session vers l'objet actif représentant le groupe distant.
Dans un second temps, quand l'appel de méthode arrivera sur le groupe actif, le gestionnaire de sécurité va établir une session vers tous les membres du groupe. Bien entendu, chacun de ces membres pourra être soit un objet actif, soit un groupe hiérarchique, soit encore une fois un groupe actif.

6.4.4 Communications de groupe multi-clusters sécurisées

Les groupes actifs sont particulièrement adaptés à la communication sécurisée entre clusters. En combinant les groupes hiérarchiques et les groupes actifs, il est possible d'obtenir un groupe dont les communications sont sécurisées pour les communications entre les divers sites et dont les communications internes à chaque site s'effectuent en clair. Ce mécanisme est présenté au sein de la figure 6.8. En effet, la communication entre l'objet appelant et le groupe actif se compose d'un seul message. Le transfert de ce message se fera selon les paramètres de sécurité négociés lors de l'établissement de la session. Une fois le message transféré au groupe actif, il sera expédié à tous les membres du groupe.

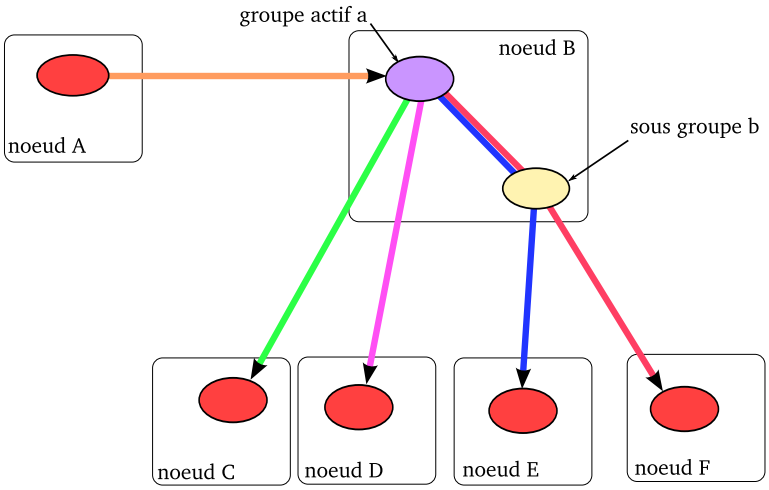


FIG. 6.7 – Groupe actif et sécurité

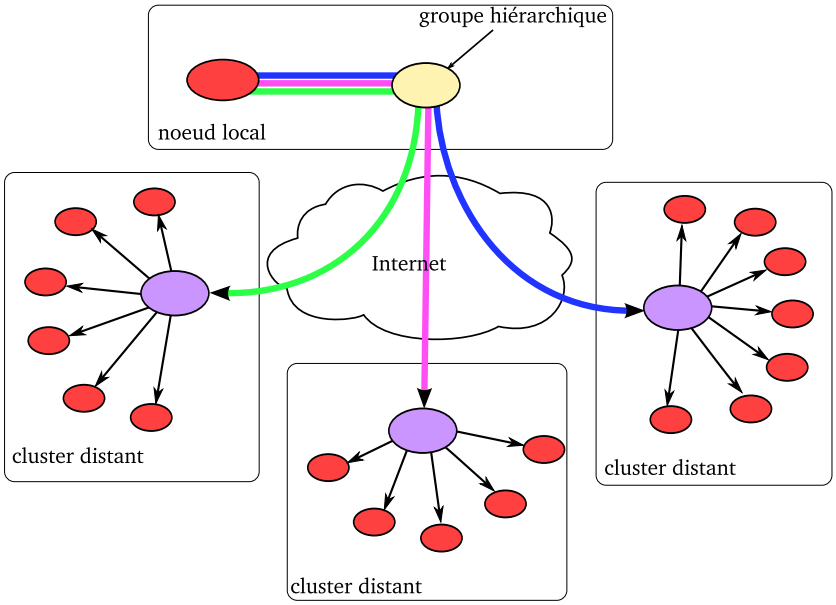


FIG. 6.8 – Communications de groupe sécurisées multi-clusters

Conclusion

En conclusion, nous avons ici un comportement différent selon le type de groupe existant. Le cas d'un groupe hiérarchique et activé se révèle adapté à la grille. Il permet la communication intercluster simple et sécurisée.

6.5 Architecture pair-à-pair et sécurité

Une application complexe est généralement distribuée afin de répartir la charge de calcul sur un nombre important d'ordinateurs. La distribution se fait généralement sur des clusters. Cette solution, utilisée depuis une dizaine d'années, comporte quelques inconvénients, tel le fait que les machines soient dédiées uniquement à un certain type de calcul, que l'accès à ces clusters soit réservé à certains utilisateurs ou encore que les communications inter-clusters soient difficiles voire impossibles. D'un autre côté, l'équipement informatique des entreprises et des particuliers n'est guère différent de l'ordinateur composant un cluster. Ces équipements sont souvent sous-exploités voire inexploités pendant de longues périodes. Partant de ce constat, il semble intéressant d'utiliser ces machines pour déployer des applications distribuées. L'organisation de ces machines au sein d'un réseau pair-à-pair permettrait de mettre la puissance inexploitée de ces machines à la disposition d'utilisateurs. Dans un tel réseau, chaque pair connaît un ensemble de voisins sans pour autant connaître la totalité des pairs du réseau (figure 6.9).

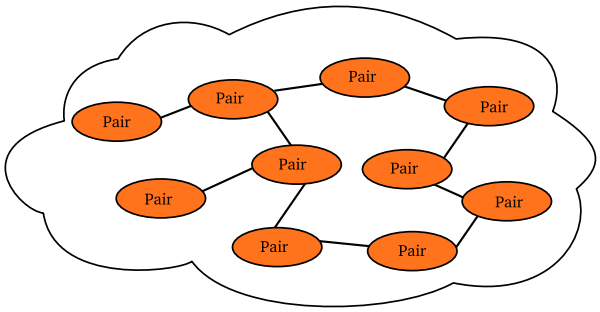


FIG. 6.9 – Réseau pair-à-pair

De manière générale, une application utilisant un réseau pair-à-pair soumet à chacun de ses pairs une succession de tâches à effectuer ; ces tâches pouvant être indépendantes ou non les unes des autres. Les recherches concernant la sécurité des applications au sein d'un réseau pair-à-pair supposent généralement la présence d'un environnement hostile. Il n'est pas possible dans ces conditions de faire confiance à l'hôte et, dans ce cas, l'utilisation de mécanismes de sécurité à base d'algorithmes cryptographiques se révèlent inutiles. S'ils permettent d'assurer l'authentification des entités en présence, la confidentialité et l'intégrité des données transmises et reçues, rien ne peut être prouvé concernant les actions effectuées par l'hôte distant.

Dans la littérature, on retrouve essentiellement deux approches qui se basent sur une vérification logicielle des résultats :

- *l'approche par un processus de vérification des résultats* (*simple checker*) [119] suppose qu'il est possible de valider la correction d'un résultat. Cependant, il semble peu probable de trouver des post-conditions permettant cette vérification pour tout type d'applications ;
- *l'approche par duplication* [108] consiste à faire exécuter plusieurs fois la même tâche à plusieurs pairs différents et de comparer les résultats. La duplication de toutes les tâches étant coûteux en temps, des modèles probabilistes [67, 52] ont été proposés afin de réduire ce coût.

L'inconvénient majeur de ces approches par rapport à nos buts réside dans le fait qu'à la fois les données et les algorithmes utilisés pour ce genre de calcul vont être facilement accessibles à n'importe quel attaquant.

Il est nécessaire de rappeler ici que la validité de notre mécanisme de sécurité peut être mise en péril si les hôtes ne sont pas de confiance. Un hôte malicieux pourrait accéder aux certificats et clés privées d'une entité et compromettre la sécurité de l'application. De toutes les ressources disponibles via le réseau pair-à-pair, seules celles qui de part leur identité et leur localisation seront autorisées par les politiques du ou des gestionnaires de sécurité pourront être utilisées par l'application.

L'architecture pair-à-pair [30], développée au-dessus de la bibliothèque *ProActive*, permet la mise à disposition de nœuds, donc de puissance de calcul, aux applications qui en ont besoin. L'infrastructure du réseau est dite à auto-organisation continue et paramétrée. La construction et la maintenance de ce réseau repose sur un jeu d'échanges de messages entre les divers pairs. Chaque pair du réseau est un objet actif qui possède une liste de nœuds qu'il peut mettre à la disposition des applications utilisant le réseau pair-à-pair. Selon sa puissance, une machine peut mettre un ou plusieurs nœuds à disposition.

Cette architecture nous permet de tester notre modèle de sécurité dans un environnement qui possède les caractéristiques suivantes :

- l'environnement est dynamique. Les ressources dont va disposer l'application vont varier au cours du temps. De nouvelles ressources vont pouvoir être acquises pendant que d'autres disparaissent, il est également possible d'acquérir à nouveau des ressources qui avaient disparues ;
- la nature des ressources (runtimes, nœuds, objet actifs) ainsi que leur localisation sont variées. Du point de vue de la sécurité, on se retrouve avec des ressources externes, sans lien direct avec l'identité de l'application. Chacune de ces ressources possède sa propre identité et ses propres règles de sécurité. Nous nous trouvons dans la cas d'un environnement où la gestion de la sécurité peut être considérée comme décentralisée.

Un serveur pair-à-pair est composé de plusieurs objets actifs :

- le `P2PService` est l'objet actif principal d'un pair. Il créé les autres objets actifs . Il sert les requêtes d'enregistrement ou les demandes de ressources.
- le `P2PNodeManager` gère les nœuds partagés par le pair qu'il représente. Il gère également le système de réservation des nœuds.
- le `P2PAcqManager` permet de maintenir à jour les connexions avec les autres pairs.
- le `P2PNodeLookup` sert d'intermédiaire lorsque le `P2PService` demande un ensemble de nœuds.
- le `FirstContact` sert lors du démarrage du pair. Il permet d'initier les premiers contacts avec les autres pairs du réseau.

Le `P2PService` étant un objet actif, il est possible de le transformer en objet actif sécurisé.

```
// chargement d'un gestionnaire de securite
ProActiveSecurityManager psm = new ProActiveSecurityManager("p2p.xml");

// creation des meta objets du futur objet actif
MetaObjectFactory factory = ProActiveMetaObjectFactory.newInstance();

// ajout du gestionnaire de securite aux meta objets
factory.setProActiveSecurityManager(psm);

// P2PService Active Object Creation
P2PService p2pService = (P2PService) ProActive.newActive(P2PService.class.getName
    (), null, url, null, factory );
```

Grâce au mécanisme de propagation dynamique du contexte de sécurité, les objets actifs créés par le `P2PService` héritent du contexte de sécurité de leur créateur. Ce contexte sera également propagé aux nœuds créés par le `P2PNodeManager`.

Ainsi, les ressources disponibles dans un réseau pair-à-pair sont vues par le mécanisme de sé-

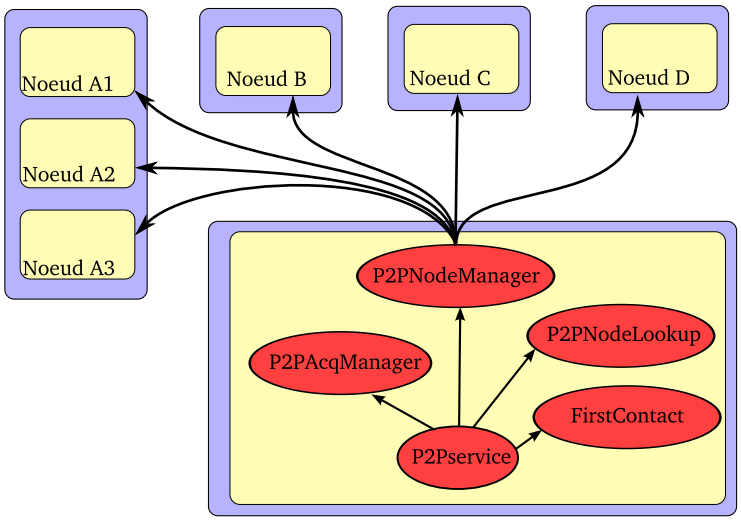


FIG. 6.10 – Un serveur pair-à-pair partageant des nœuds

curité comme des entités sécurisées normales possédant leurs propres politiques de sécurité. Lors du déploiement de l'application, il est possible de créer un nœud sur un runtime obtenu par le biais du réseau pair-à-pair. Le nœud sera associé à la politique de sécurité de l'entité qui l'a créé. Cependant, il est aussi possible d'acquérir un nœud existant et possédant sa propre politique de sécurité. Un objet actif créé sur un tel nœud sera soumis non seulement à la politique de sécurité de l'application qui l'a déployé mais aussi à la politique de sécurité du nœud qui l'accueille. Le fait que la référence sur un runtime ait été acquise via le réseau pair-à-pair ou obtenue lors du déploiement de l'application est transparente vis-à-vis du mécanisme de sécurité, seule compte la politique de sécurité du runtime.

Lorsqu'une application utilise le mécanisme pair-à-pair et la sécurité, deux paramètres peuvent influer sur le temps de calcul global : les communications et la volatilité des pairs. En effet, la surcharge occasionnée par notre modèle de sécurité est essentiellement concentrée au niveau des communications et lors de la création d'entités sécurisées.

Le calcul pair-à-pair est particulièrement bien adapté aux applications demandant un ratio (Temps de communication / Temps de calcul) très petit. Cette propriété est intéressante, elle nous laisse supposer que la surcharge induite par le chiffrement des communications n'aura qu'un faible impact sur l'application.

La volatilité des pairs est plus critique, elle implique la génération d'un nouveau certificat pour chaque nouveau pair. Pour avoir une idée du coût de la surcharge liée au mécanisme de sécurité, le temps de génération du certificat doit être comparé à la durée de vie du pair. Intuitivement, on déduit que le coût de la surcharge diminuera avec l'augmentation de la durée de vie du pair. Ainsi une application qui crée un grand nombre de pairs de très courte durée se verra plus pénalisée qu'une dont la durée de vie des pairs est plus grande.

Notre but est maintenant de montrer le comportement du mécanisme de sécurité dans le cas d'une application écrite spécifiquement pour l'architecture pair-à-pair. Cette application se penche sur le problème des n reines qui doit répondre à la question suivante : "Comment placer n reines sur un échiquier de $n \times n$ sans qu'elles soient en prises ?". Notons pour la petite histoire que la bibliothèque *ProActive* a permis d'établir un nouveau record mondial le 11 juin 2005 en calculant les solutions possibles pour n=25. Ce résultat a impliqué en moyenne un total de 260 machines de bureau pendant près de 6 mois [93].

L'application est de type maître-esclaves et se compose de trois parties :
- *NQueensManager* : c'est le maître (serveur). Il s'occupe de la création des *Workers* sur les nœuds mis à sa disposition, ainsi que de la distribution des tâches à ces derniers. Il collecte également les résultats transmis par les *Workers*.
- *Worker* : c'est l'esclave (client). Il reçoit une tâche du *NQueensManager*, la calcule et retourne le résultat. Dès qu'il a fini son travail le *NQueensManager* lui renvoie une nouvelle tâche.
- *Task* : le problème est découpé de manière statique en tâches. Ces tâches sont affectées à des workers libres.

Le *NQueensManager* et les *Workers* sont des objets actifs. Il est possible traiter en parallèle autant de tâches qu'il y a de *Workers*.

Si on suppose que le *NQueensManager* a été lancé avec une politique de sécurité, alors :
- le *NQueensManager* ne pourra créer des *Workers* seulement sur les nœuds autorisés par la politique de sécurité ;
- les *Workers* étant créés par le *NQueensManager*, ils appartiendront à la même application que le *NQueensManager* (leur certificat dérivant du certificat d'application du *NQueensManager*), ils posséderont la même politique de sécurité que le *NQueensManager*. Tout ceci est réalisé implicitement par le mécanisme de propagation dynamique du contexte de sécurité ;
- les attributs de sécurité d'une communication entre le *NQueensManager* et un *Worker* dépendront de leurs localisations (négociation dynamique de la politique de sécurité).

Les points exposés ci-dessus montrent que le mécanisme de sécurité est capable de s'adapter à une application dont les ressources varient dynamiquement au cours de son exécution. Le mécanisme de sécurité est capable de supporter une architecture pair-à-pair (mais les hôtes doivent être de confiance). Cependant, cette application étant du type maître-esclaves, tous ses pairs ne sont pas égaux, si la disparation d'un *Worker* n'a que peu de conséquences, la disparation du *NQueensManager* entraîne irrémédiablement la fin du calcul.

Le nombre total de tâches calculées a été de 12 1251 99 et la durée moyenne d'une tâche de 140 secondes. Si on considère qu'un certificat a été créé pour chaque tâche et que le temps de génération de ce certificat est de une seconde (voir section 7.1) alors le coût induit par la sécurité sur la tâche est de 0,7%.

Une application est considérée comme *pure pair-à-pair* [110] si on peut enlever un pair sans que cela n'entraîne de dégradation importante sur l'ensemble de l'application. Nous allons maintenant réutiliser l'exemple des N-reines dans une version qui correspond mieux cette définition.

Le but de l'application est de mettre en évidence le comportement du mécanisme de sécurité dans un contexte pur pair-à-pair. Les hypothèses et l'architecture de l'application sont les suivantes :
- l'ensemble des tâches à effectuer est connu, les tâches sont au préalable découpées et stockées à un endroit accessible par les *Workers*. Pour simplifier, nous supposerons qu'il s'agit ici d'un ensemble d'objets actifs nommés *TasksServers*. Leur localisation est connue et fixe. Ces *TasksServers* sont regroupés au sein d'un même nœud virtuel nommé *TasksServersVN*.
- le comportement d'un *Worker* est le suivant :

```
debut
  tant que (des taches sont disponibles)
     Si (des pairs sont disponibles) alors
          créer au maximum deux nouveaux workers
     finsi
     récupérer une tâche à effectuer depuis un serveur
     calculer la tâche
     envoyer le résultat au serveur
  fin tant que
```

```
fin
```

Chaque worker est inclus dans un nœud appartenant au nœud virtuel *WorkerVN*.

– Seuls les *Workers* ayant le même certificat d'application que les *TasksServers* pourront télécharger des tâches et déposer des résultats.

Au démarrage, un *Worker* sécurisé va commencer par créer deux autres *Workers*. Grâce au mécanisme implicite de propagation dynamique du contexte de sécurité, chacun des deux nouveaux *Workers* aura un gestionnaire de sécurité qui contiendra la politique de sécurité de l'application ainsi qu'un certificat dérivé du certificat d'application. Ils disposeront ainsi des mêmes possibilités (création de nouveaux *Workers* et téléchargement des tâches) que leur parent. Les avantages de cette approche décentralisée sont doubles :

– Elle permet une croissance exponentielle du nombre de *Workers*,
– mais surtout de distribuer efficacement le coût de la génération des certificats sur tous les *Workers*.

L'utilisation de l'architecture pair-à-pair a permis de tester notre mécanisme de sécurité sur un système dont la gestion de la sécurité peut être considérée comme décentralisée et dynamique. Les exemples nous ont permis de montrer les réactions du mécanisme de propagation dynamique du contexte de sécurité couplé avec la combinaison des politiques de sécurité au sein d'un environnement pair-à-pair. L'analyse des divers cas survenant lors de l'utilisation du système pair-à-pair montre que notre modèle de sécurité peut être utilisé dans un environnement dynamique.

6.6 Tolérance aux pannes et sécurité

L'ensemble des diverses ressources utilisables par une application utilisant la bibliothèque est vaste. Il est possible de déployer une application sur une machine multiprocesseur, sur des grilles de calcul ou encore sur des réseaux ad-hoc créés à partir d'ordinateurs de bureau. Toutes ces ressources ne disposent pas des mêmes taux de fiabilité. Ainsi, il est plus probable de voir disparaître un ordinateur de bureau suite à son redémarrage par son utilisateur principal pour une raison donnée ou sa déconnexion du réseau que de perdre une machine appartenant à une grille de calcul pour ces mêmes raisons. Ainsi, du fait de la nature volatile des ressources, une application peut perdre une partie d'elle même et de ses données suite à l'extinction d'une machine donnée. Dans ce contexte, l'utilisation d'un protocole de tolérance aux pannes se révèle plus que nécessaire afin de palier à la volatilité des ressources et de permettre de minimiser l'impact de la perte d'une partie de l'application sur le temps de calcul global de l'application.

Il est intéressant de noter qu'il n'existe pas qu'un seul type de panne mais plusieurs ; elles sont classées en quatre catégories majeures [70] que nous présentons brièvement :

– *les pannes franches* (*crash, fail-stop*), que l'on appelle aussi arrêt sur défaillance. C'est le cas le plus simple : on considère qu'un processus peut être dans deux états, soit il fonctionne et donne le résultat correct, soit il ne fait rien. Il s'agit du type de panne actuellement supporté par le mécanisme de tolérance aux pannes de *ProActive* ;
– *les pannes par omission* (*transient, omission failures*). Dans ce cas, on considère que le système peut perdre des messages. Ce modèle peut servir à représenter des défaillances du réseau plutôt que des processus ;
– *les pannes de temporisation* (*timing, performance failures*). Ce sont les comportements anormaux par rapport à un temps, comme par exemple l'expiration d'un délai de garde ;
– *les pannes arbitraires*, ou byzantines (*malicious, byzantine failures*). Cette classe représente toutes les autres pannes : le processus peut alors faire “n'importe quoi”, y compris avoir un comportement malveillant.

Le protocole de tolérance aux pannes intégré dans la bibliothèque se classe parmi les protocoles de points de reprise induits par message, avec reprise asynchrone [11]. Son fonctionnement est le suivant : chaque objet actif prend périodiquement un point de reprise. Ce point de reprise est numéroté dans l'ordre croissant. Chaque message émis par un objet actif est annoté du nu-

méro du dernier point de reprise. Cette approche suppose la présence d'un support de stockage stable accessible par tous les objets de l'application. Ce support est présent sous la forme d' un serveur qui centralise les points de reprise des divers objets de l'application : le *serveur de checkpoint* (figure 6.11). Il reçoit les informations en provenance de tous les objets actifs de l'application qu'il surveille et a la charge de redémarrer les objets actifs en panne dans leur dernier état stable connu si un problème survient lors de l'exécution.

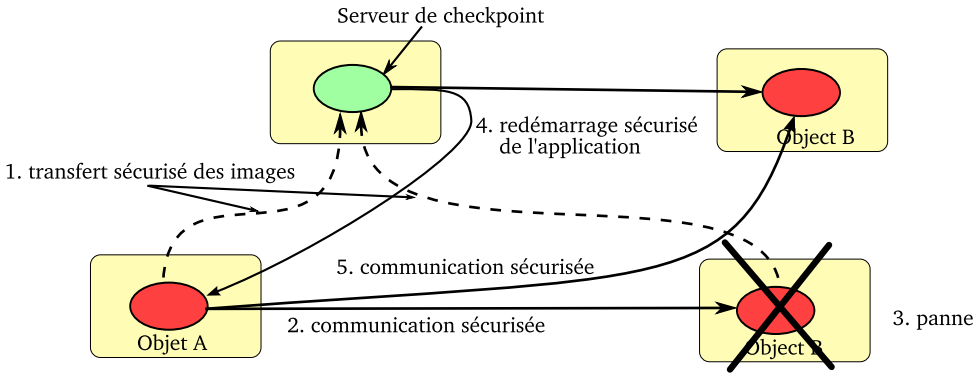


FIG. 6.11 – Tolérance aux pannes et sécurité

Nous allons maintenant étudier la réaction du protocole de tolérance face à une panne franche et les implications vis-à-vis du mécanisme de sécurité.

Dans un premier temps, intéressons nous aux communications existantes. Les objets actifs doivent envoyer une image de leur état au serveur de checkpoint. Dans le cas où ce serveur est un objet actif, cette communication se résume à un appel de méthode depuis un objet actif vers un autre. Nous avons déjà vu que le mécanisme de sécurité est capable d'intercepter et d'appliquer les attributs de sécurité nécessaires à une telle communication. La sécurisation des communications entre un objet actif et son serveur de checkpoint se fait de manière transparente. Le serveur de checkpoint reçoit les images et les stocke sans avoir conscience du protocole de sécurité sous-jacent.

Lors d'une panne, le serveur de checkpoint doit redémarrer l'application (tous les objets) au dernier état stable connu, y compris l'objet actif en panne. Son redémarrage se fera sur un nœud différent de son nœud initial. La création des objets actifs est dépendante de l'autorisation que possède le serveur de checkpoint vis-à-vis de la création d'un objet actif sur le nœud de secours. Comme dans le cas de la création d'un objet actif par un autre, cette autorisation est tributaire d'une part des politiques de sécurité des entités contenant le serveur et de celle du serveur et, d'autre part, des politiques de sécurité des entités contenant le nœud cible ainsi que la politique de ce dernier. De plus, le transfert de l'image se fera de manière sécurisée comme n'importe quelle création d'objet actif sur un runtime distant.

La deuxième étape lors du redémarrage d'un objet actif consiste à ré-émettre un certain nombre de messages (requêtes et réponses) émis avant la panne et désignés comme devant être réémis par le protocole de tolérance aux pannes. Ces messages sont conservés de manière non chiffrée au sein de l'image de l'objet actif. Au moment de leur ré-émission, ces messages le seront selon la politique de sécurité de l'objet actif et des entités englobantes en fonction de sa nouvelle localisation et de la nouvelle localisation de l'objet cible dans le cas où il aurait été redémarré sur un autre nœud que son nœud initial.

L'analyse de ce scénario de redémarrage de l'application utilisant à la fois le protocole de tolérance aux pannes et l'architecture de sécurité montre qu'il est possible de combiner ces deux fonctionnalités. Il est néanmoins nécessaire de donner au serveur de checkpoint les droits et au-

torisations nécessaires qui lui seront demandés lors du redémarrage de l'application. Dans le cas contraire, l'application ne pourrait être redémarrée même si le mécanisme de tolérance aux pannes aurait été en mesure de la redémarrer.

Dans la taxonomie des pannes que nous avons présenté précédemment, il est intéressant de se pencher également sur le cas des pannes byzantines. Il est admis que, dans ce type de panne, le processus défaillant peut-être amené à exécuter n'importe quelle action et notamment acquérir un comportement malveillant. Dans notre cas, ce processus malveillant est représenté par un objet actif. Si on considère que la panne n'affecte que le comportement métier de l'objet actif et pas son gestionnaire de sécurité qui appartient au niveau méta, alors ce dernier est capable de contenir certains des agissements de l'objet actif malveillant selon la politique de sécurité de l'application et dans la limite des interactions gérées par un gestionnaire de sécurité. Effectivement, le comportement malveillant ne peut être contenu s'il s'agit d'actions non perceptibles par le gestionnaire de sécurité (occupation totale du processeur ou de la mémoire, écriture sur le disque, modification des données internes de l'objet). Cette limitation n'est pas dû à notre modèle mais plutôt à son implantation au-dessus de la machine virtuelle qui n'offre pas les fonctionnalités et/ou les informations nécessaires pour surveiller et contrôler finement l'exécution d'un processus java (*thread*).

Par exemple, considérons qu'un objet actif ait été déconnecté du reste de son application et que suite à cette déconnexion, il ait acquis un comportement malveillant qui consiste à :
- scanner tous les registres disponibles à la recherche d'objets actifs ;
- une fois un objet actif trouvé, il essaie dans un premier temps de communiquer avec cet objet puis de migrer vers cet objet.

Dans ce cas, nous nous trouvons face à des interactions sur lesquelles le gestionnaire de sécurité peut intervenir. Ce dernier, comme dans le cas standard, n'autorisera l'objet actif à communiquer ou à migrer que selon la politique qu'il aura établi en fonction de l'objet actif cible.

Pour conclure, nous avons montré dans cette section qu'il était possible de combiner deux comportements non fonctionnels (tolérance aux pannes et mécanisme de sécurité) présents au sein de la bibliothèque. Cette combinaison est possible notamment grâce à la propriété de transparence que possède le mécanisme de sécurité vis-à-vis des autres fonctionnalités, son utilisation étant implicite lors de l'utilisation de communications entre objets actifs.

6.7 Modèle à composants hiérarchiques et distribués

La conception, le développement et la maintenance d'applications distribuées à large échelle confrontent développeurs et administrateurs à des contraintes de réutilisation de code existants dans des contextes différents, de configurations de logiciels et matériels dans des contextes matériels et logiciels amenés à évoluer au cours du temps. La programmation de ces applications a été influencée par divers modèles tels que la programmation modulaire, la programmation objet et finalement l'utilisation de composants logiciels. Il n'existe pas une définition universelle du composant logiciel, cependant une définition généralement admise est celle donnée par J. Harris en 1995 :

Définition 13 *Composant logiciel :*
Un composant est un morceau de logiciel assez petit pour qu'on puisse le créer et le maintenir, et assez grand pour qu'on puisse l'installer et en assurer le support. De plus, il doit être doté d'interfaces standards pour pouvoir interopérer avec des entités tierces.

Dans son approche de bibliothèque pour la construction d'applications pour les grilles de calcul, *ProActive* propose un modèle de composants basé sur le modèle de composants *Fractal* [20]. Le modèle à composant Fractal fournit une vision homogène du logiciel en définissant un ensemble réduit de concepts tels que composants, contrôleur, contenu, interface et binding. Sa principale innovation réside dans la définition récursive de la définition de la structure d'un composant.

Définition 14 *composants ProActive :*
Un composant est formé d'un ou plusieurs objets actifs s'exécutant sur un ou plusieurs runtimes. Un composant peut être configuré en utilisant un fichier descripteur définissant les diverses parties du composant au travers d'un langage de description d'architectures (ADL) [12].

Un composant est destiné à être utilisé dans des applications distribuées et parallèles doit pouvoir exploiter ces concepts. Pour cela, il existe trois types de composants bien spécifiques (figure 6.12) : primitif, composite ou parallèle.

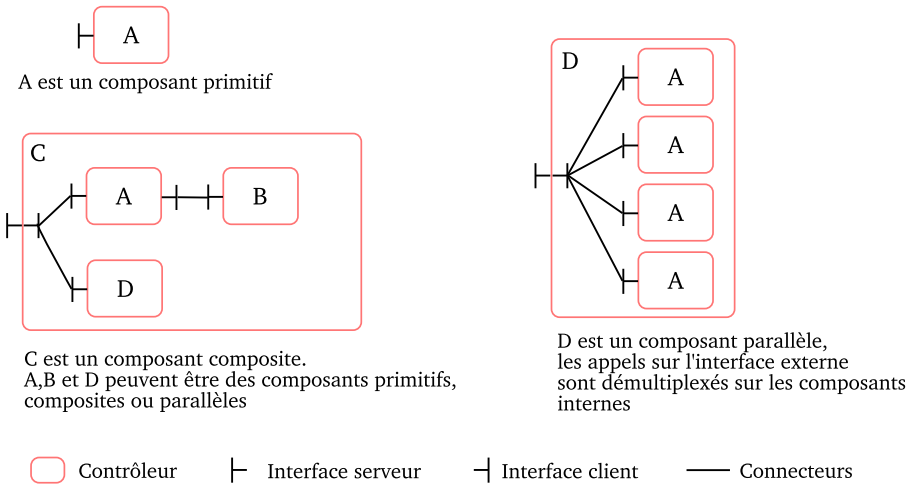


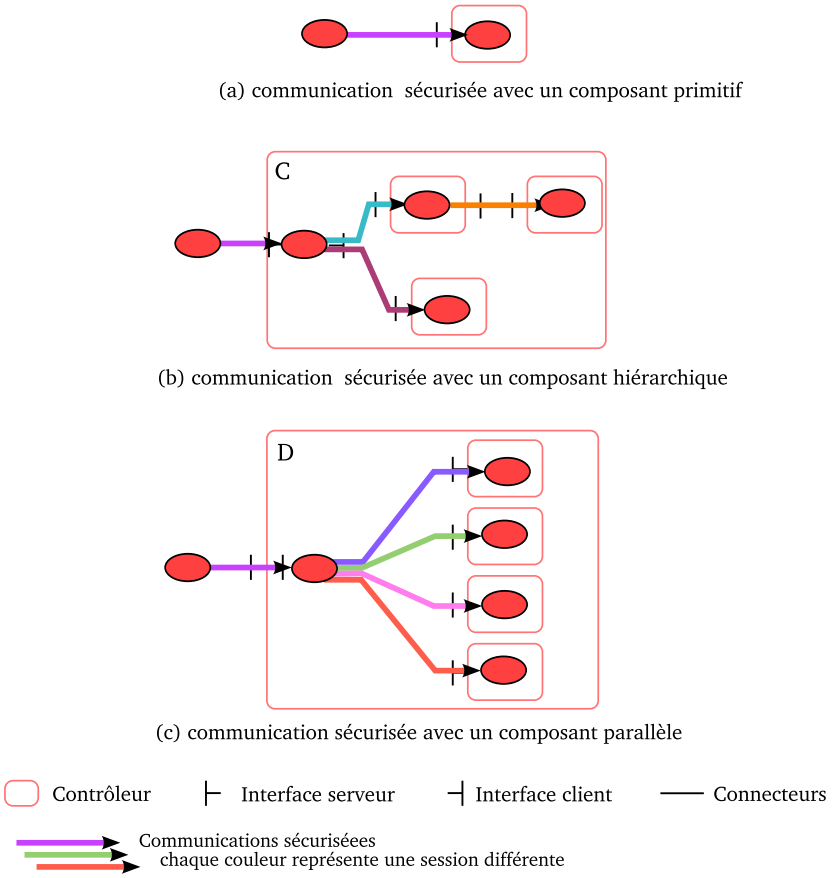
FIG. 6.12 – Divers types de composants

La définition récursive du composant du fait de l'utilisation du modèle Fractal permet de combiner ces trois types entre eux afin de former des composants adaptés aux grilles de calcul. Ainsi, les éléments d'un composant parallèle peuvent être distribués sur les machines d'une grille de calcul. Comme tout objet accessible à distance, un composant possède des besoins en termes d'authentification des composants tiers, de confidentialité et d'intégrité des communications. L'aspect hiérarchique de la définition pose aussi les mêmes problèmes mais concernant les communications d'un composant hiérarchique avec les composants qui le composent.

Un composant primitif, composite ou parallèle, étant un objet actif possède un `Body` et des méta-objets, notamment un gestionnaire de sécurité. De ce fait, un composant est considéré comme un objet actif à part entière du point de vue du mécanisme de sécurité.

De ce fait, les communications entre un objet actif et un composant ou encore entre deux composants peuvent être sécurisées grâce à notre mécanisme de sécurité. Le schéma (a) de la figure 6.13 présente une configuration contenant un objet actif effectuant un appel sur un composant primitif. On s'aperçoit que l'appel traverse de manière transparente l'enveloppe du composant pour aller directement à l'objet actif que représente le composant. Il en est de même pour le mécanisme de sécurité qui accède directement au gestionnaire de sécurité du composant (donc de l'objet actif) pour échanger les paramètres de sécurité nécessaires pour l'établissement de la session.

Le schéma (b) de la figure 6.13 présente un appel de méthode depuis un objet actif vers un composant composite. Comme dans le cas précédent, une session est établie entre l'objet appelant et le composant composite C. Lorsque le composant C reçoit l'appel de méthode, il le répercute vers un ou plusieurs de ses composants internes selon la méthode appelée. Le composant C établira, via son gestionnaire de sécurité, une session avec le ou les composants sur lesquels l'appel doit être continué. De même, si un composant interne doit faire un appel à un autre composant, le gestionnaire de sécurité établira au besoin une session vers ce dernier avant d'autoriser l'appel de méthode.



(a) communication sécurisée avec un composant primitif

(b) communication sécurisée avec un composant hiérarchique

(c) communication sécurisée avec un composant parallèle

FIG. 6.13 – Communications sécurisées avec les composants

Le composant parallèle (schéma (c) figure 6.13) est construit au-dessus du mécanisme de communication de groupe. Les composants du composant parallèle sont regroupés au sein d'un groupe. La communication sécurisée au sein d'un composant parallèle se comporte de manière identique à celle d'une communication de groupe. Selon le fait que le groupe est un groupe simple, hiérarchique ou actif, le gestionnaire de sécurité s'adaptera aux propriétés des communications sécurisées au sein d'un groupe d'objets actifs telles que présentées dans la section 6.4.

6.8 Sémantique des communications sécurisées

6.8.1 Communications locales et distantes

Dans la partie 4.3, nous avons présenté la sémantique des communications telle qu'elle existe au sein de la bibliothèque. Nous allons reprendre les points et montrer comment l'introduction de la sécurité s'intègre avec le modèle existant.

Lorsque deux objets actifs se trouvent dans le même espace d'adressage (i.e. dans la même machine virtuelle), ils doivent pouvoir communiquer directement sans passer par la pile des protocoles réseaux, tout en conservant une sémantique identique. Cette optimisation existe dans *ProActive* cependant elle maintient la sémantique de passage de paramètres par copie profonde, garantissant ainsi l'absence d'objets passifs partagés.

Au niveau du protocole de sécurité, on peut se demander si, lors d'un appel de méthode local, le chiffrement de la communication n'est pas superflu vu l'inexistence du risque d'interception de la communication par une tierce partie. Cependant, nous avons fait le choix de chiffrer les communications même à l'intérieur du même espace adressage. Si, du point de vue des performances, cette solution n'est pas optimale, elle a l'avantage de garantir la sémantique du mécanisme de sécurité : toute communication qu'elle soit locale ou distante sera chiffrée si la politique de sécurité l'impose.

6.8.2 Sécurisation des messages

Nous allons présenter ici les techniques mises en œuvre pour appliquer le principe de transparence de la sécurité par rapport au code application. Un des effets de bord positifs de cette approche est d'étendre la transparence de la sécurité bien au delà du code applicatif et de d'implanter la transparence dans les méta-objets eux-mêmes.

Pour cela, nous avons étendu l'interface *Message* qui est l'interface mère de tous les messages échangés entre les diverses entités composant une application. La principale modification a fait en sorte que le message puisse gérer lui-même sa sécurité. En effet, lorsqu'un message va être envoyé sur le réseau, il doit être sérialisé. Un objet sérialisable est capable de gérer lui-même sa propre sérialisation. À partir de ce moment, le message est capable de détecter qu'il va être envoyé sur le réseau. Il va ainsi pouvoir sécuriser les diverses données sensibles qu'il contient en fonction de la politique de sécurité qui aura été négociée par le gestionnaire de sécurité.

Pour chaque classe implantant l'interface `Message`, il est nécessaire de spécialiser les méthodes gérant la sécurisation du message. Effectivement, selon le type de la classe, les données sensibles ne seront pas les mêmes. Par exemple, la classe `RequestImpl` contient une variable d'instance correspondant à l'appel de méthode et à ses paramètres. Cette variable n'existe bien évidemment pas dans la classe `ReplyImpl` qui contient une autre variable représentant le résultat d'un précédent appel de méthode.

6.8.3 Appel de méthode sécurisé

La figure 6.14 présente les divers objets et méta-objets traversés lors d'un appel de méthode sécurisé d'un objet actif sur un autre.

On s'aperçoit en premier lieu que le chemin suivi par l'appel de méthode sécurisé est ressemble fortement à celui d'un appel de méthode non sécurisé. On a rajouté en fait deux indirections supplémentaires. La première se situe au moment de l'envoi du message vers le body

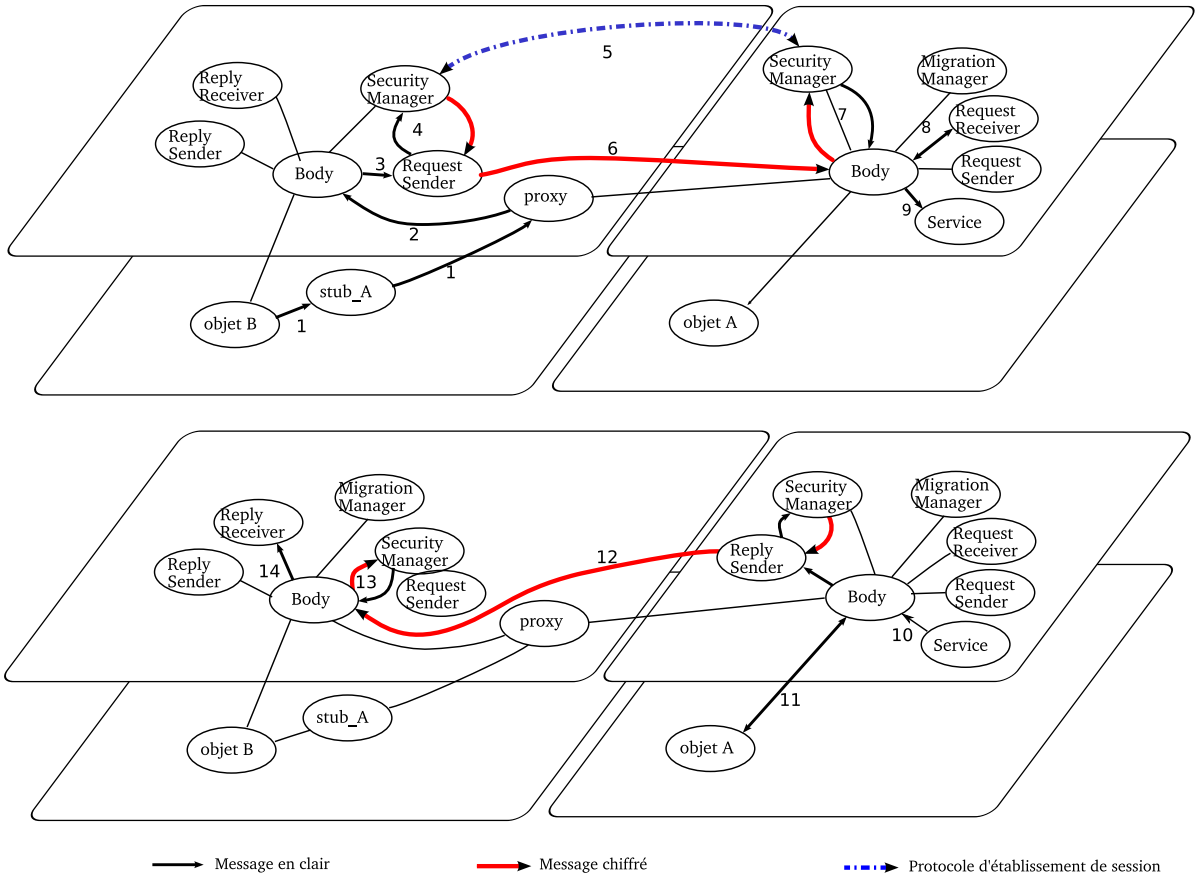


FIG. 6.14 – Appel de méthode sécurisé

distant. C'est en effet à ce moment de l'appel de méthode que le message allant être sérialisé contacte le gestionnaire de sécurité afin que ce dernier établisse une session avec l'objet cible (étape 5). Une fois la session établie, le message peut être chiffré et expédié à l'objet cible (étape 6). Une fois reçu, le message est dirigé vers le gestionnaire de sécurité de l'objet cible pour être déchiffré. Nous sommes toujours dans la phase de *rendez-vous* (voir section 4.3) ce qui implique que si une erreur survient dans le déchiffrement ou la validation du message, l'objet appelant peut en être informé et le cas échéant prendre la décision de réémettre le message. Une fois, déchiffré et validé le message est mis dans la queue des requêtes en attente d'être servi par l'objet actif. Un mécanisme similaire est mis en place pour l'envoi des réponses.

6.8.4 Type d'attaques

Nous allons maintenant nous intéresser aux types d'attaques possibles. On supposera que la politique de sécurité associée au message attaqué est prévue pour supporter le type d'attaque. Une politique activant les trois attributs de communication activés protégera contre l'ensemble des attaques ci-après. Un message reçu par une entité sécurisée est transmis aux autres méta-objets seulement s'il est conforme aux attributs de sécurité requis par la session qui lui est associée et validé par les mécanismes cryptographiques associés.

Interception

Un tiers non autorisé intercepte des données. Il s'agit d'une attaque sur la confidentialité.

La confidentialité des messages est protégée par un cryptosystème symétrique utilisant *AES*. La clé de session est échangée lors de la première interaction des deux entités sécurisées en présence et reste valable jusqu'à la fin de la durée de validité de la session. La taille de la clé symétrique est de 192 bits.

L'attaquant ne peut lire le contenu du message qu'à la condition de découvrir la clé de session.

Modification

Un tiers non autorisé intercepte des données et les modifie avant de les envoyer au destinataire. Il s'agit d'une attaque sur l'intégrité.

L'intégrité des messages est assurée par un algorithme *HMAC* [66] utilisant SHA-1 pour calculer l'empreinte du message. Un code HMAC est utilisé pour déterminer si un message transmis sur un canal non sécurisé a été falsifié, à condition que l'expéditeur et le destinataire partagent une clé secrète. Cette clé secrète est créée lors de l'initialisation de la session.

L'attaquant ne peut modifier le contenu du message de manière transparente sans connaître la clé utilisée pour générer l'empreinte. Si l'empreinte ne correspond pas au code HMAC calculé par le receveur, le message est ignoré.

Rejeu

L'attaquant qui a réussi à intercepter des messages les ré-émet dans le but d'obtenir des informations ou de perturber la cible de l'attaque. Nous considérons qu'il s'agit d'une attaque sur l'intégrité des messages.
Le mécanisme mis en place pour contrer ce type d'attaque consiste à inclure le code HMAC du message précédent lors de génération du code HMAC du message courant. Cette approche permet de créer une chaîne de messages, la validation du code HMAC d'un message rejoué échouera.

Fabrication

Un tiers non autorisé insère des données contrefaites dans les communications de l'application. Il s'agit d'une attaque sur l'authentification.

Pour qu'un message soit accepté, il doit être validé par une session, c'est-à-dire être déclaré comme authentique par les mécanismes cryptographiques associés à la session. Avant d'envoyer un message entièrement fabriqué, l'attaquant doit découvrir la politique de sécurité de la session qu'il veut attaquer. Si on considère qu'il a été capable d'intercepter un message, il peut analyser ce dernier et regarder si le message est chiffré ou si une signature est présente. Il peut à partir de là deviner la politique de sécurité de la session. Cependant, à moins de détenir les secrets nécessaires à la fabrication du message, il sera incapable d'émettre un message qui sera validé par les mécanismes cryptographiques de sa cible.

Usurpation d'identité

Le protocole d'échange de clé *Diffie-Hellman* que nous utilisons pour l'échange de la clé de session est connu pour être vulnérable à l'attaque dite d'impersonnation. Cet algorithme ne se base en effet que sur les clés publiques des entités en présence pour échanger la clé de session.

Dans notre cas, ce protocole est complété par une authentification à base de certificat rendant impossible ce type d'attaque lorsque l'authentification est requise par la politique de sécurité.

Interruption

Une partie de l'application distribuée est détruite ou est devenue inaccessible. Il s'agit d'une attaque sur la disponibilité.

Ce type d'attaque n'est pas géré directement par le mécanisme de sécurité mais par le mécanisme de rendez-vous. Ce dernier garantit la délivrance du message lors d'une communication. Si le rendez-vous échoue, le comportement actuel est de lever une exception afin d'en informer l'utilisateur.

6.9 Conclusion

Dans ce chapitre, nous nous sommes attachés à exposer l'intégration de notre modèle de sécurité au sein de la bibliothèque *ProActive* ainsi que ses interactions avec les autres fonctionnalités (fonctionnelles ou non) de la bibliothèque.

La première partie de ce chapitre a permis une présentation générale de la façon dont s'intègre notre modèle de sécurité au sein de la bibliothèque. L'organisation de la bibliothèque en couches distinctes nous a permis d'ajouter la couche liée à la sécurité juste au dessus de la couche transport. Cet ajout s'est fait de manière transparente vis-à-vis des couches voisines qui n'ont pas conscience de l'existence de cette couche intermédiaire. Nous avons également détaillé pour chaque entité le fonctionnement du mécanisme de sécurité en présentant le code java qui aurait du être ajouté au code métier de l'application sans l'approche implicite pronée par notre mécanisme de sécurité. Nous avons également exposé l'interface de programmation disponible dans le cas où le programmeur voudrait, au sein de son application, interagir avec le mécanisme de sécurité.

La deuxième partie de ce chapitre a présenté l'intégration du mécanisme de sécurité avec les fonctionnalités majeures de la bibliothèque.

Tout d'abord, nous nous sommes intéressés à la migration des activités et aux implications de cette mobilité sur notre modèle de sécurité. Pour chacune des implications et seulement dans les cas nécessaires, nous avons présenté une solution permettant de la rendre compatible avec notre modèle de sécurité.

La deuxième fonctionnalité majeure présente dans la bibliothèque est la communication de groupe. Il existe plusieurs types de groupe chacun avec des propriétés propres. L'implantation des groupes peut être vue comme une extension d'une communication *ProActive* normale. De ce

fait, les groupes héritent directement de la sécurité qui était déjà intégrée dans les communications standards.

L'architecture Pair-à-Pair ajoute la découverte dynamique d'hôtes (runtimes) ou de nœuds à la bibliothèque. Une telle architecture peut être considérée comme totalement décentralisée et dynamique. Nous avons ainsi pu montrer la capacité de notre modèle de sécurité à fonctionner sans difficulté au sein d'un tel système. Le mécanisme de propagation dynamique du contexte de sécurité se révèle être la partie du modèle qui apporte la flexibilité nécessaire lors de la création de nouvelles entités sur des ressources acquises via le système Pair-à-Pair.

Nous nous sommes ensuite intéressés au mécanisme de tolérance aux pannes. La gestion de gestion de ce dernier est implanté selon le même principe de transparence que notre modèle de sécurité, l'intégration des deux mécanismes fut relativement aisée.

La dernière fonctionnalité à profiter du mécanisme de sécurité implicite que nous avons décrite est le modèle à composant de *ProActive*. Ce modèle repose à la fois sur les fonctionnalités de base de la bibliothèque mais aussi sur les communications de groupe. Il hérite lui aussi du mécanisme de sécurité que nous avons implanté au sein de la bibliothèque.

Finalement, nous avons exposé la sémantique des communications sécurisées. Les communications entre les entités sécurisées sont composées d'un échange de messages. Ces messages peuvent être expédiés au moyen de plusieurs protocoles de transport. Pour ces raisons, nous avons tout naturellement implanté une sécurité au niveau message plutôt qu'au niveau transport. Il est ainsi possible d'ajouter de nouveaux protocoles de transport sur lesquels des messages sécurisés pourront être expédiés.

Ainsi, nous avons pu, tout au long de ce chapitre, mettre en évidence les avantages d'utiliser un mécanisme de sécurité implicite. Il permet de créer des applications génériques qui pourront être déployées avec ou sans sécurité selon le contexte. Il simplifie également l'intégration des mécanismes de sécurité avec les autres mécanismes offerts par une bibliothèque.

Chapitre 7

Tests et performances

7.1 Tests unitaires

Cette première section nous permet de présenter les temps d'initialisation nécessaires aux divers protocoles sur lesquels reposent notre solution de sécurité. Ces tests sont effectués sur des pentium IV à 3.2Ghz, 1Go RAM, sous linux fedora core 1 doté d'un kernel 2.4.22. Nous utilisons la JVM de Sun version 1.5.0 ainsi que la bibliothèque de sécurité `Bouncycastle` 1.20. Les machines sont interconnectées par un réseau ethernet 100Mbits.

La première série de tests (tableau 7.1) nous permet de mettre en évidence la durée de génération d'un certificat pour une entité sécurisée en fonction de la taille de la clé RSA associée au certificat (algorithme à clé publique, chiffrement asymétrique).

Le choix des diverses tailles de la clé RSA s'est effectué par rapport à l'article présenté dans la section 2.4. Nous rappelons, selon l'article, il est possible de calculer la taille de clé RSA en fonction de l'année courante lors de sa génération. Le respect de cette équivalence permet de protéger la confidentialité des données sur une période de 20 ans.

Nous avons mesuré la durée de génération d'une paire de clés RSA grâce à la bibliothèque `Bouncycastle` ainsi que la génération d'une paire de clés RSA par `OpenSSL`. `OpenSSL` est une bibliothèque implantant le protocole TLS et SSL. Dans de telles conditions, il apparaît qu'une

Année	Taille de la clé RSA d'une entité	Temps (Java)	Temps (OpenSSL 0.9.7a)
2000	512	118 ms	35 ms
2002	1 024	460 ms	120 ms
2005	1 149	906 ms	226 ms
2010	1 369	1 900 ms	285 ms
2023	2 054	6 000 ms	980 ms
2050	4 047	26 000 ms	9 000 ms

TAB. 7.1 – Temps de création d'une entité selon la taille de la clé RSA

application voulant garantir un niveau de sécurité maximal se doit de générer des entités sécurisées en accord avec la limite théorique pour 2005 soit une taille de clé de 1149 bits. Il faudra 900 ms pour générer la paire de clés RSA ainsi que le certificat de l'entité. Ce temps de génération est à rapprocher du temps de création d'un objet actif sans sécurité qui est de l'ordre de 80 ms.

La deuxième série de tests (tableau 7.2) s'intéresse au temps mis pour générer une clé de session AES (chiffrement symétrique, utilisé dans les échanges de messages) en fonction de la taille de cette dernière et toujours en fonction de la limite théorique. Nous rappelons qu'une clé de session est générée entre deux entités lors de leur première interaction si la politique de sécurité le requiert.

Finalement, nous nous sommes intéressés au coût du mécanisme de sécurité lors d'un appel

Année	Taille de la clé de session	Temps
2000	70	0.5 ms
2002	72	0.5 ms
2005	74	0.6 ms
2010	78	0.6 ms
2023	88	0.6 ms
2050	109	0.7 ms

TAB. 7.2 – Temps de génération d'une clé de session

de méthode. La figure 7.1 présente la durée d'un appel de méthode entre deux objets actifs en fonction de la taille des paramètres de l'appel. Nous avons fait varier la taille des paramètres de 10 kilo-octets à 50 méga-octets. La communication sécurisée s'effectue avec les attributs de confidentialité et d'intégrité activés. Un appel de méthode sécurisé consiste à successivement calculer la signature des paramètres, chiffrer les données, les transmettre sur le réseau, déchiffrer les données et finalement calculer leur signature afin de la comparer à celle reçue. Cette succession de mécanismes de sécurité est coûteuse en temps de calcul et est fonction de la taille des données à traiter, ce qui explique la divergence des deux courbes.

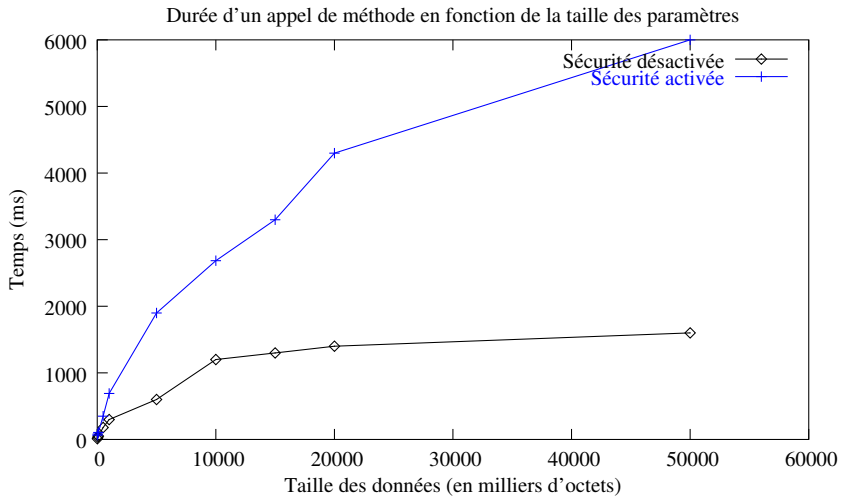


FIG. 7.1 – Durée d'un appel de méthode en fonction de la taille des paramètres

7.2 Les itérations de Jacobi

On s'intéresse ici à la résolution d'un système linéaire de la forme $Ax = b$ où A est la matrice de taille $n * m$, $\forall n, m \in \mathbb{N}$.

L'application utilisée dans cette section permet la mise en œuvre de l'algorithme des itérations de Jacobi au sein d'une application distribuée. Sa particularité réside dans l'utilisation des communications de groupes et du modèle de programmation OO-SPMD présent dans la bibliothèque *ProActive*. Cette application fait partie des exemples de programmation distribuée présents au sein de la bibliothèque.

Nous avons choisi cet exemple car cet algorithme implique des communications entre chaque sous-matrice voisine à chaque itération. Nous sommes en présence d'une application qui communique souvent, nous allons pouvoir nous intéresser à l'impact du mécanisme de sécurité sur le temps total d'exécution de l'application.

Les évaluations de performances suivantes ont été réalisées sur l'infrastructure proposée par le projet *Grid 5000*. Ce projet, financé par l'ACI GRID, a pour but de fournir une plate-forme de grilles regroupant 8 sites géographiquement distribués en France et regroupant 5000 processeurs. Les grappes sont interconnectées via Renater avec une bande passante de 2,5 Gb/s. L'application a été déployée sur la grappe de Nice regroupant des ordinateurs bi-AMD Opteron 64 bits à 2 Ghz, 2 Go de RAM. Le système d'exploitation est Red Hat 9 doté d'un kernel 2.4.21. Les machines sont interconnectées par un réseau Ethernet 1 Gb/s. Nous utilisons la JVM de Sun version 1.5.0 ainsi que la bibliothèque de sécurité `Bouncycastle` 1.20.

7.2.1 Algorithme

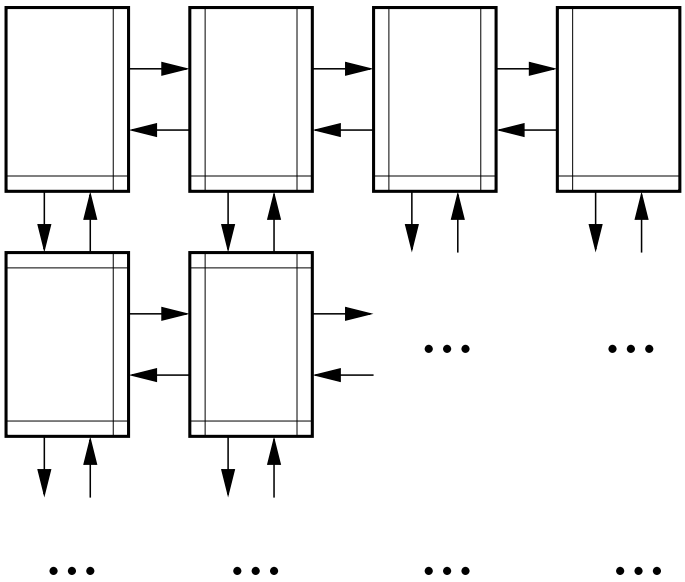


FIG. 7.2 – Algorithme distribué

La matrice initiale est découpée en x sous-matrices de taille $n' * m'$. Chaque sous-matrice est représentée par un objet actif. À chaque itération de l'algorithme, la valeur d'un point d'une sous-matrice est remplacée par la moyenne de ses voisins du nord, sud, est et ouest (figure 7.2). Les points sur les frontières des sous-matrices sont échangés avec la sous-matrice voisine. La valeur des données sur les frontières de la matrice initiale ne change pas au cours du temps.

Nous présentons la boucle principale de l'algorithme. `neighbors` représente le groupe de sous-matrices.

```
me = ProActive.getStubOnThis();
public void jacobiIteration() {
  internal_compute(); //updates converged
  neighbors.send(boundariesGroup);
  ProSPMD.barrier({"send", ... ,"send"});
  me.boundaries_compute(); //updates converged
  me.exchange();
  if (!converged) me.jacobiIteration();
}
```

L'algorithme utilisé pour les mesures de performances et son implémentation ont fait l'objet d'une publication [6].

7.2.2 Architecture abstraite

L'architecture de déploiement est des plus simples. On retrouve le nœud local qui sert au déploiement de l'application. Ce nœud appartient à un nœud virtuel *Launcher*. Le second nœud

virtuel *matrixNode* héberge les nœuds contenant les solveurs. Un solveur est un object actif qui est en charge d'une sous-matrice. Le nombre de sous-matrices dépend directement du nombre de nœuds disponibles dans le nœud virtuel *MatrixNode*.

7.2.3 Déploiement

La politique décrite est constituée d'une seule règle qui impose que toutes les communications soient authentifiées, chiffrées et que leur intégrité soient vérifiée. Ce cas représente bien évidemment le pire cas au niveau du surcoût induit au niveau des performances. La politique de sécurité associée au déploiement sécurisé de l'application est la suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy>
 <ApplicationName>JacobiIteration</ApplicationName>
 <Certificate>jacobi.pkcs12</Certificate>
 <Rules>
  <Rule>
      <From>
       <Entity type="Default" name=""/>
      </From>
      <To>
       <Entity type="Default" name=""/>
      </To>
      <Communication>
      <Outgoing value="authorized">
         <Attributes authentication="required" integrity="required"
             confidentiality="required"/>
      </Outgoing>
      <Incoming value="authorized">
         <Attributes authentication="required" integrity="required"
             confidentiality="required"/>
      </Incoming>
      </Communication>
      <Migration value="authorized"/>
      <OACreation value="authorized"/>
      <Request value="authorized"/>
      <Reply value="authorized"/>
   </Rule>
 </Rules>
</Policy>
```

7.2.4 Analyse des résultats

Nous avons évalué les performances de l'algorithme des itérations de Jacobi avec et sans le mécanisme de sécurité activé en faisant varier la taille des matrices de 2 millions de doubles à 25 millions de doubles.

Pour chaque test, la matrice est découpée en 16 (4x4) sous-matrices. Chacune d'elles est représentée par un objet actif. On a placé deux objets actifs par ordinateur. Le nombre de connexions que va établir un objet actif dépend de sa position dans la matrice. Un objet actif représentant un coin de la matrice ne possède que deux parents proches avec qui il communiquera tandis qu'un objet actif se trouvant au milieu de la matrice communiquera avec ses quatre parents. Le nombre d'objets `Session` qui représentent une communication sécurisée est identique au nombre de communications que l'objet actif établira.

La figure du haut représente la durée de l'initialisation des matrices en fonction de la taille de la matrice initiale. Elle représente le temps mis par l'application pour découper et transférer les données aux huit ordinateurs. La figure du bas représente la durée moyenne d'une itération, en

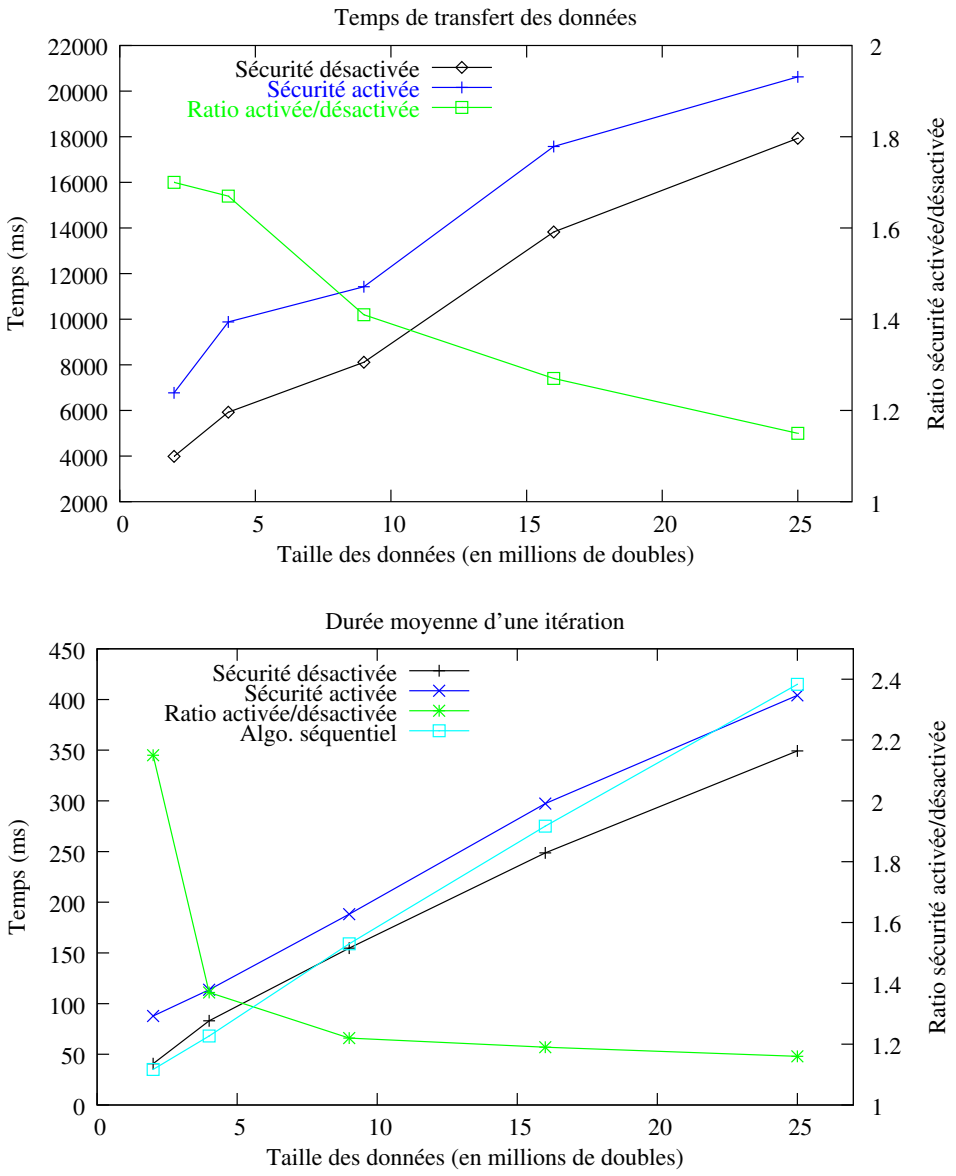


FIG. 7.3 – Jacobi : Découpage de la matrice sur huit ordinateurs

millisecondes, en fonction de la taille des données. La quatrième courbe *Algo. séquentiel* présente les performances de la version mono-processeur et séquentielle de l'algorithme.

Nous avons décidé d'évaluer les performances du mécanismes de sécurité sur ces deux points car les tous dépendent des communications effectuées, soit l'endroit où le mécanisme de sécurité intervient principalement.

Les tests montrent que dès que la taille des données augmente, le pourcentage du coût induit par les protocoles de sécurité baisse rapidement. Sur la figure représentant le temps de calcul (bas), on passe d'un pourcentage allant de 120% pour une matrice ne contenant que 2 millions de doubles à un pourcentage atteignant seulement 18% lorsque la taille de la matrice atteint 25 millions de doubles.

La chute brutale du ratio du coût de la sécurité en début s'explique par le temps constant que requiert l'initialisation d'une session sécurisée et notamment des objets de cryptographie de la bibliothèque Java.

L'analyse des courbes représentant les ratios montrent que le ratio tend à se stabiliser avec l'augmentation de la taille des données et donc du temps de calcul global de l'application. Dès lors, le mécanisme de chiffrement induit un coût constant qui semble se maintenir aux alentours de 18%.

On constate également que la durée d'une itération de la version sécurisée, distribuée sur huit processeurs est égale à celle de la version séquentielle pour un taille de matrice d'environ 22 millions de doubles. A partir de ce point, la version sécurisée et distribuée est plus rapide que la version séquentielle. Pae exemple, lorsque la taille des données atteint 25 millions de doubles, une itération de la version sécurisée est plus rapide de 4% qu'une itération de la version séquentielle.

Chapitre 8

Conclusion

Ce chapitre résume la thèse et présente des perspectives de recherche ouvertes à la suite de nos travaux de doctorat.

8.1 Bilan et Contributions

Nous nous sommes intéressés à la création d'un modèle de sécurité dont le fonctionnement est implicite vis-à-vis de l'application et indépendant du code métier de celle-ci. Nous nous sommes particulièrement attachés aux applications réparties et aux problèmes de sécurité survenant lors de leur exécution. Un des buts d'une application répartie de type grille est d'utiliser le maximum de ressources disponibles afin de terminer un calcul le plus rapidement possible. Nous avons vu que l'ajout de mécanismes de sécurité au sein du code source d'une application rendait particulièrement difficile sa conception, son évolutivité et sa généricité. Le problème vient du fait que cette approche standard mélange deux concepts orthogonaux. D'un côté nous avons la partie fonctionnelle de l'application qui est présente sous la forme de code métier, de l'autre côté nous trouvons les mécanismes de sécurité qui, s'ils sont indéniablement nécessaires lors de l'utilisation d'applications réparties sur des réseaux publics, se trouvent être des concepts orthogonaux au code d'une application. Pour cette raison, nous nous sommes appliqués à rendre notre modèle le plus transparent possible.

Les applications cibles de notre modèle étant les applications réparties, nous nous sommes efforcés de créer un modèle qui supporte à la fois, les contraintes et les besoins de ce type d'application. Notre objectif était de proposer un modèle de sécurité dont l'utilisation serait la plus simple possible pour tous les acteurs pouvant intervenir dans le déroulement des différentes phases d'une application répartie.

Après avoir présenté, dans un premier temps, les principes de sécurité existants (chapitre 2), nous nous sommes intéressés aux mécanismes de sécurité présents au sein de plusieurs intergiciels intervenant dans la construction d'applications réparties. Cette étude nous a permis de déterminer l'adéquation de ces intergiciels avec notre approche de gestion transparente des fonctionnalités de sécurité.

En se basant sur ces constats, notre approche a consisté à étudier le fonctionnement d'une application afin d'identifier les interactions devant être prises en compte par le mécanisme de sécurité. Cette étape a permis de découpler au maximum la partie fonctionnelle d'une application de sa partie non fonctionnelle. Nous avons ensuite intégré au sein de la bibliothèque *ProActive* qui a servi de support à notre modèle, les modifications nécessaires pour intégrer notre modèle de sécurité. Ainsi, nous avons pu retirer du code applicatif toutes les notions de sécurité en ne laissant à la vue du développeur ou de l'utilisateur que des mécanismes simples à mettre en œuvre et à utiliser.

Les apports de cette thèse sont :

– **La définition d'un modèle de sécurité hiérarchique : les entités sécurisées.** L'or-

ganisation du modèle en entités sécurisées nous a apporté une gestion hiérarchique des politiques de sécurité. Il nous a été possible de spécifier des contraintes sur la sécurité des échanges entre deux de ces entités. La définition générique d'une entité sécurisée, nous apporte la flexibilité nécessaire pour appliquer ce concept aux divers objets ayant la capacité, à la fois, d'être accessible à distance mais aussi de pouvoir communiquer avec des objets distants.

- **une intégration implicite des mécanismes de sécurité** : L'utilisation d'un protocole à méta-objet a permis l'implantation de l'intelligence du mécanisme en charge de la sécurité dans le niveau méta soit en dehors du contexte fonctionnel de l'application. Grâce à cette indirection, il nous a été possible de rajouter tout le mécanisme de sécurité nécessaire dans le niveau méta sans avoir à modifier ni le code source, ni le code compilé de l'application. Ce mécanisme peut dynamiquement prendre les décisions relatives à la sécurité en fonction du contexte dans lequel il se trouve. La configuration de ces mécanismes au sein de fichiers externes a permis de les rendre adaptables à toute situation.

- **La définition d'un langage de politique de sécurité** qui permet d'exprimer les règles de sécurité de notre modèle de sécurité. De part ses propriétés. notamment son adaptabilité ce langage permet une création et un gestion décentralisées des politiques de sécurité. Il permet également d'exprimer des règles concernant la combinaison des plusieurs politiques de sécurité.

- **L'implantation du modèle au sein de la bibliothèque *ProActive* .** Elle a permis de mettre en évidence la faisabilité du modèle ainsi que son adaptabilité lors de son intégration avec les autres fonctionnalités de la bibliothèque.

D'un point de vue technique, le code du mécanisme de sécurité comporte toutes les fonctionnalités nécessaires à son fonctionnement. Son architecture interne, flexible, permet d'imaginer l'utilisation de nouvelles ou futures notions liées à la sécurité. Son impact restreint sur les autres méta objets de la librairie permettra son intégration dans les fonctionnalités que pourrait proposer la bibliothèque *ProActive* . Les travaux présentés dans cette thèse ont donné lieu aux publications suivantes [3, 4, 5, 7] dans des conférences internationales ainsi qu'à la rédaction d'un chapitre de livre [8] sur le thème du grid computing.

8.2 Perspectives

Notre approche nous permet à la fois de garantir les besoins en terme de sécurité mais aussi de mettre à disposition des programmeurs et des utilisateurs un ensemble de fonctionnalités leur permettant une gestion de haut-niveau de la sécurité de leurs applications. Même si le modèle est suffisant pour être utilisé au sein d'applications distribuées, il est indéniable que certains concepts présents dans les autres intergiciels pourraient être intégrés dans notre modèle. Il est aussi possible d'envisager l'évolution de certains de nos principes afin d'améliorer notre modèle de sécurité. Nous abordons ici les axes de recherches qu'il reste à explorer ou approfondir.

8.2.1 Modèle de sécurité

- *Support du contrôle de flux*
  Le modèle de sécurité dans sa version actuelle permet de sécuriser des communications point-à-point entre diverses entités sécurisées mais ne place aucune hypothèse sur les données. En effet, une donnée transmise par une entité A à une entité B dépendra par la suite de la politique de sécurité de l'entité B et non plus de celle de l'entité A.
  Si nous supposons que ces deux entités appartiennent au même domaine de sécurité (i.e.

même application), nous pouvons supposer que la politique de sécurité de l'application a été établie de manière cohérente afin de ne pas introduire de faille de sécurité au sein de l'application. Cependant, notre modèle permet également à des entités sécurisées contenues dans des contextes de sécurité différents d'interagir. Dans ce cas précis, une fois la donnée transmise, elle peut être utilisée par l'entité qui l'a reçue sans aucune contrainte. Elle ainsi être transmise à une autre entité lors d'un appel de méthode. Il est ainsi possible d'exporter des données en dehors de leur contexte de sécurité initial si la politique de sécurité le permet. Au vu de ces explications, il parait intéressant d'étudier l'adjonction d'un mécanisme de contrôle de flux afin combler cette potentielle brèche de sécurité. Les premières réflexions sur cette possible extension nous ont conduit essayer de reprendre le principe des entités sécurisées afin de transformer les données en entités sécurisées. Cette solution est similaire à l'approche des *Secure Meta Objects* introduite en 2.7.4.
Si cette solution semble prometteuse, elle comporte néanmoins des inconvénients. Tout d'abord, elle va augmenter considérablement le nombre d'objets puisque nous comptons associer une entité sécurisée à toute donnée expédiée lors d'un appel de méthode. Deuxièmement, cette approche suppose qu'à tout instant, le moniteur de référence est capable de calculer la politique de sécurité associée à une donnée. Ceci suppose que chaque donnée soit associée à une ou plusieurs politiques de sécurité. Enfin, il existe une limitation venant directement du langage Java et qui concerne les types primitifs. Jusqu'en Java 1.4, ces types ne pouvaient être réifiés, dès lors il nous est impossible d'y associer une entité sécurisée. Depuis la version 1.5, Java introduit les mécanismes d'*autoboxing* permettant la conversion automatique d'un type primitif vers un objet `wrapper` représentant un entier et inversement. Cependant il ne nous semble pas possible de conserver les méta-objets associés à un objet lors de sa conversion en type primitif.

– *Reconfiguration dynamique des entités sécurisées*
Nous avons vu qu'une entité sécurisée est associé à une politique de sécurité lors de sa création. Afin de permettre à cette entité sécurisée de réagir à la dynamicité de son environnement, il faut être capable de modifier dynamiquement la politique de sécurité associée à l'entité. Par exemple, lorsque l'entité sécurisée fournit un service, il n'est pas concevable de devoir arrêter puis recréer le service lors de changement dans la politique de sécurité. La mise à jour de la politique de sécurité d'une entité est d'une mise en œuvre assez simple. Cependant, la mise à jour de la politique de sécurité d'une entité sécurisée peut avoir des répercussions surtout si cette entité appartient à une application distribuée. Dans le cadre d'une application sécurisée, nous avons fait en sorte que toutes les entités sécurisées partagent la même politique de sécurité afin de garantir une politique de sécurité cohérente pour l'application. Ainsi, la mise à jour d'une entité devrait être répercutée sur toutes les entités composant l'application. Le problème principal consiste donc à mettre à jour toutes ces entités en même temps ce qui suppose d'avoir un mécanisme de recherche et/ou de localisation.

– *Service web sécurisé*
L'exportation d'un objet actif en tant que service web permet à ce dernier d'être contacté par des objets par n'importe quel type de processus et pas seulement des objets actifs. De plus, le standard des services web définit ses propres mécanismes de sécurité *WS-Security*. Une perspective serait d'établir une passerelle entre notre modèle de sécurité et celui des services web. Elle permettrait de faire communiquer un service web sécurisé avec un service web représentant un objet actif et inversement. Le service web sécurisé devra interagir directement avec les mécanismes de sécurité des objets actifs et s'intégrer de manière transparente avec le mécanisme de sécurité afin de proposer un ensemble de sécurité homogène et cohérent. Ce modèle doit assurer la sécurité des messages de et vers un objet actif en se basant sur les standards utilisés au sein de la bibliothèque *ProActive* et des web services. Le deuxième objectif serait l'importation automatique d'un service web sécurisé au sein d'un code écrit avec *ProActive* . Le service web sera vu comme un objet actif au sein du code. Pour cela, il faudra écrire un mécanisme de *wrapper* qui respecte la sémantique de la bibliothèque tout en établissant un lien sécurisé vers le service web.

– *Optimisation de la surcharge cryptographique*
Notre solution permet de prendre en compte les besoins en terme de sécurité des divers acteurs intervenant dans l'élaboration de la politique de sécurité. Cette approche permet d'éviter de chiffrer plusieurs fois de suite des flux et évite ainsi une perte de performance lors des communications. L'approche actuelle est indépendante du protocole réseau sous-jacent. Cette fonctionnalité peut être à double tranchant. N'ayant pas d'informations sur le réseau, il se peut que le chiffrement effectué par notre mécanisme de sécurité soit à son tour chiffré par des mécanismes de plus bas niveau.

– *Extension du protocole de déploiement*
Le déploiement via un descripteur de sécurité pose le problème de l'activation et de la configuration de sécurité à appliquer au runtime distant. Il faut copier les descripteurs de déploiement, le fichier de règles de sécurité et, au besoin, les fichiers représentant les certificats des entités listées dans le fichier de politique de sécurité. Une solution serait la mise en place de mécanismes au niveau du descripteur de déploiement qui permettraient de gérer l'installation distante des fichiers de sécurité requis.

– *Utilisation d'un cryptosystème à courbe elliptique*
La génération d'une paire de clés publique/privée pour le cryptosystème RSA est coûteuse en temps du fait qu'elle requiert des clés de grande taille. La cryptographie à courbe elliptique [65, 28] est une branche prometteuse de la cryptographie à clé publique. Elle offre potentiellement la même sécurité que les système de cryptographie à clé publique mais avec des clés de tailles réduites. A titre de comparaison, une taille de clé de 160 bits en utilisant les courbes elliptiques équivaut en terme de niveau de sécurité à une clé RSA de 1024 bits.

8.2.2 Vers d'autres mondes

Plates-formes de services ouvertes

Les applications dont l'architecture peut évoluer dynamiquement en cours d'exécution sont un domaine de recherche en vogue actuellement. Le besoin principal vient du fait que certaines applications sont sensibles à leur contexte ou peuvent avoir besoin de charger/mettre à jour/décharger certains de leurs modules sans pour autant pouvoir interrompre leur exécution. La spécification *Open Service Gateway Interface* (OSGi) [97] se présente actuellement comme le standard de fait pour ces plates-formes. *Eclipse*, l'interface de développement GPL développée initialement par IBM, a remplacé son système de *plug-ins* par des services OSGi permettant à l'application d'évoluer, de se reconfigurer en temps réel sans redémarrage.

L'architecture d'une plate-forme OSGi (figure 8.1) se compose d'une plate-forme de services

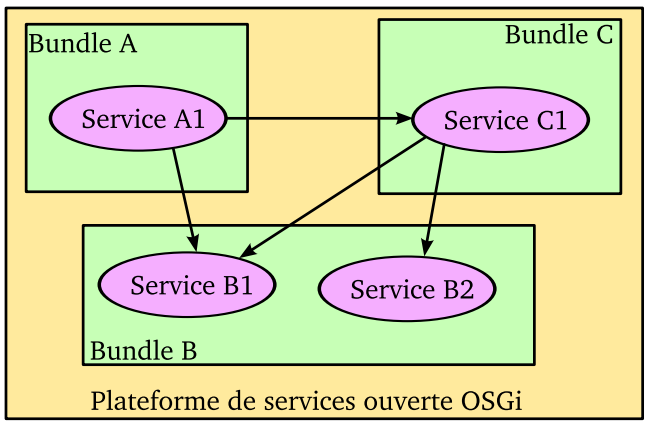


FIG. 8.1 – Architecture OSGi

ouverte ; elle représente l'environnement logiciel permettant de gérer le cycle de vie d'autres applications. Au sein de cette plate-forme, des *bundles* qui représentent des espaces de services, permettent d'exposer un certain nombre de services.

Actuellement, la spécification OGSi ne possède pas de notion concernant la distribution des services. Les recherches visant à introduire cette notion de distribution n'en sont encore qu'à leurs débuts. Il serait intéressant d'étudier les fonctionnalités que peut apporter notre mécanisme de sécurité au sein d'une telle plate-forme. Comme première approche permettant de visualiser ce qui pourrait être fait, nous pouvons établir un parallèle entre les divers éléments d'une infrastructure OSGi et une infrastructure *ProActive* . Le runtime correspondrait à la plate-forme d'accueil OSGi, les applications aux *bundles* et les objets actifs aux services.

Environnement ubiquitaire

Ce manuscrit a permis de présenter un nouveau modèle d'infrastructure de sécurité (les entités sécurisées) et leur intégration au sein de la bibliothèque *ProActive* . Cependant, nous pensons que les configurations qui ont donné naissance à ce modèle peuvent se retrouver dans d'autres environnement que les applications distribuées.

Nous pensons tout particulièrement aux *Smart Devices* et aux *Pervasive networks*. L'infrastructure de ces environnements se compose d'un ensemble d'éléments hétérogènes (capteurs, moteurs, ordinateurs) interagissant entre eux afin de mener à bien leurs missions. Les réseaux de capteurs, par exemple, sont des architectures ouvertes dans lesquelles n'importe quel intrus peut potentiellement intercepter les communications ou encore émettre de faux messages. Afin de garantir ces communications, il est nécessaire d'utiliser des protocoles cryptographiques. Chaque capteur doit pouvoir être identifié ainsi que les messages qu'il émet. Le principal problème de sécurité consiste en la génération, la diffusion ou encore l'invalidation des clés ou des certificats.

Il serait intéressant d'étudier l'adéquation de notre modèle de sécurité avec un réseau de capteurs et notamment notre façon d'attribuer des certificats aux divers capteurs. On peut, en effet, mettre en avant une similitude, au niveau conceptuel, entre les objets actifs et les capteurs. Chacun dans leur domaine, ces unités de base doivent coopérer afin de réaliser une tâche précise.

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    = DNS_Name | IP
 String     = Alpha String | Number String
 Alpha      = [a-z,A-Z]
 IPNumber   = [0-255]
 Number     = [0-9]

 RULES          =   PERMISSION* MIGRATION* OACREATION*
 PERMISSION     =   DomainName UNI_BI  DomainName ":" COM_MODE
 MIGRATION      =   DomainName "migration" FROM_TO DomainName : YES_NO COM_MODE
 OACREATION     =   DomainName "create on" DomainName : YES_NO COM_MODE
 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
                    "migration"   #rules for migration
FROM_TO         =   "->"  |
                    "<-"
YES_NO          =   "YES"  |
                    "NO"
```

Bibliographie

[1] C. Adams. *The Simple Public-Key GSS-API Mechanism (SPKM)*, October 1996. RFC 2025.

[2] Ameneh Alireza, Ulrich Lang, Marios Padelis, Rudolf Schreiner, and Markus Schumacher. The challenges of corba security. In *Workshop Sicherheit in Mediendaten, Gesellschaft für Informatik (GI)*, pages 61–72, September 2000.

[3] Isabelle Attali, Denis Caromel, and Arnaud Contes. Hierarchical and Declarative Security for Grid Applications. In *Proceedings of the International Conference On High Performance Computing*, Hyderabad, India, December 2003. Springer Verlag.

[4] Isabelle Attali, Denis Caromel, and Arnaud Contes. Une architecture de sécurité déclarative et hiérarchique pour les grilles de calcul. In INRIA, editor, *2ème rencontre francophone sur le thème Sécurité et Architecture Réseau*, pages 203–212, Nancy, France, July 2003.

[5] Isabelle Attali, Denis Caromel, and Arnaud Contes. Deployment-Based Security for Grid Applications. In *The International Conference on Computational Science (ICCS 2005), Atlanta, USA, May 22-25*, LNCS. Springer Verlag, 2005.

[6] Laurent Baduel, Françoise Baude, and Denis Caromel. Object-Oriented SPMD with Active Object. In *Proceedings of Cluster Computing and Grid*, Cardiff, United Kingdom, May 2005.

[7] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. Components for numerical GRIDs. Invited paper in European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS, July 2004.

[8] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, 2004.

[9] David Basin, Frank Rittinger, and Luca Viganò. A formal analysis of the corba security service. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002 : Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 330–349. Springer-Verlag, Heidelberg, 2002.

[10] Françoise Baude, Denis Caromel, Lionel Mestre, Fabrice Huet, and Julien Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[11] Francoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. A hybrid message logging-cic protocol for constrained checkpointability. In *Proceedings of EuroPar2005*, August-September 2005. to appear.

[12] Francoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*, Springer Verlag, 2003. Lecture Notes in Computer Science, LNCS.

[13] D. E. Bell and L. J. LaPadula. Secure computer system : Unified exposition and multics interpretation. Technical Report MTR-2997, The MITRE Corporation, March 1976.

[14] John K. Bennett. The design and implementation of distributed smalltalk. *ACM SIGPLAN Notices*, 22(12) :318–330, 1987.

[15] T.A. Berson. Differential cryptanalysis mod 2^{32} with applications to MD5. In R.A. Rueppel, editor, *Advances in Cryptology — Eurocrypt '92*, Berlin, 1992. Springer-Verlag.

[16] K. Biba. Integrity consideration for secure computer systems. Technical Report MTR-3153, MITRE Corporation, 1975.

[17] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. In A.J. Menezes and S. A. Vanstone, editors, *Proceedings of CRYPTO 90*, pages 2–21. Springer-Verlag, 1991. Lecture Notes in Computer Science No. 537.

[18] Bob Blakley. *CORBA Security*. Object Technology Series. Addison-Wesley, 1999.

[19] Jean-Pierre Briot and Rachid Guerraoui. Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances. *Technique et Science Informatiques (TSI)*, 15(6) :765–800, June 1996.

[20] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the Seventh International Workshop on Component-Oriented Programming at ECOOP*, Malaga, Spain, June 2002.

[21] M. Burnside, D. Clarke, T. Mills, A. Maywah, S. Devadas, and R. Rivest. Proxy-based Security Protocols in Networked Mobile Devices. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 265–272. ACM Press, 2002.

[22] Ning Chen. A case study of the ejb security : Combining declarative, role-based access control with programmatic application-specific proxy security checks. In *PDPTA '02 : Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1344–1347. CSREA Press, 2002.

[23] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents : Are they a good idea ? Technical Report RC 19887 (December 21, 1994 - Declassified March 16, 1995), IBM, Yorktown Heights, New York, 1994.

[24] David Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proc. IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[25] E. Cohen and D. Jefferson. Protection in the hydra operating system. *ACM SIGOPS*, 9(5) :141–160, November 1975.

[26] System Security Study Committee. *Computer at Risk : Safe Computing in the Information Age*. National Academy Press, 1991.

[27] H. de Meer, J-P Richter, A. Puliafito, and O. Tomarchio. Tunnel agents for enhanced internet QoS. *IEEE Concurrency*, 6(2) :30–39, 1998.

[28] N. Demytko. A new elliptic curve based analogue of RSA. volume 765 of *Lecture Notes in Computer Science*, pages 40–49. Springer-Verlag Inc., 1994.

[29] Robert H. Deng, Shailendra K. Bhonsle, Weiguo Wang, and Aurel A. Lazar. Integrating security in CORBA based object architectures. In *1995 IEEE Symposium on Security and Privacy*, pages 50–61, Oakland, CA, USA, May 1995. IEEE Comput. Soc. Press.

[30] Alexandre di Costanzo. Modèle et infrastructure de programmation pair-à-pair. Master's thesis, Nice University France, 2004.

[31] T. Dierks and C. Allen. The TLS Protocol - Version 1.0 RFC 2246, 1999.

[32] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6) :644–654, 1976.

[33] A. Diller. *An Introduction to Formal Methods*. John Wiley & Sons, 1994.

[34] Hans Dobbertin. Cryptanalysis of MD5 compress. *Lecture Notes in Computer Science*, 1039 :53–69, 1996. Fast Software Encryption Workshop.

[35] C. Ellison. SPKI Requirements, RFC 2692. Technical report, IETF, September 1999.

[36] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI Certificate Theory, RFC 2693. Technical report, IETF, September 1999.

[37] P. Eronen, J. Lehtinen, J. Zitting, and P. Nikander. Extending jini with decentralized trust management, 2000. 3rd IEEE Conference on Open Architectures and Network Programming (OPENARCH 2000), pages 25–29.

[38] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents : Authentication and state appraisal. In Elisa Bertino, Helmut Kurth, Giancarlo Martella, and Emilio Montolivo, editors, *Computer Security-ESORICS 96. 4th European Symposium on Research in Computer Security Proceedings*, number 1146 in Lecture Notes in Computer Science, pages 118–130, Rome, Italy, September 1996. Springer.

[39] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents : Issues and requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, Baltimore, Md., October 1996.

[40] D.F. Ferraiolo and D.R. Kuhn. Role based access control. In *15th National Computer Security Conference*, pages 554–563, 1992.

[41] Adam Ferrari, Frederick Knabe, Marty Humphrey, Steve Chapin, and Andrew Grimshaw. A flexible security system for metacomputing environments. Technical Report CS-98-36, 1, 1998.

[42] Adam Ferrari, Frederick Knabe, Marty Humphrey, Steve J. Chapin, and Andrew S. Grimshaw. A Flexible Security System for Metacomputing Environments. In Peter M. A. Sloot, Marian Bubak, Alfons G. Hoekstra, and Louis O. Hertzberger, editors, *HPCN Europe*, volume 1593 of *Lecture Notes in Computer Science*, pages 370–380. Springer, 1999.

[43] S. Foley. Aggregation and Separation as Noninterference Properties. *Journal of Computer Security*, 1(2) :159–188, 1992.

[44] Philip W. L. Fong. Viewer's discretion : Host security in mobile code systems. Technical Report SFU CMPT TR 1998-19, School of Computing Science, Simon Fraser University, Burnaby, BC, November 1998.

[45] I. Foster, Kesselman, C., and S. Tuecke. The anatomy of the grid : Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications, 15 (3)*, 2001.

[46] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *Int. Journal of Supercomputer Applications*, 11(2) :115–128, 1997.

[47] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid : An Open Grid Services Architecture for Distributed Systems Integration. http ://www.globus.org/research/papers/ogsa.pdf, January 2002.

[48] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5) :342–361, May 1998.

[49] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1996.

[50] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital Distributed System Security Architecture. In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 305–319, 1989.

[51] Morrie Gasser. *Building a secure computer system*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[52] Cécile Germain-Renaud and Nathalie Playez. Result checking in global computing systems. In *ICS '03 : Proceedings of the 17th annual international conference on Supercomputing*, pages 226–233, New York, NY, USA, 2003. ACM Press.

[53] C. Ghezzi and G. Vigna. Mobile Code Paradigms and Technologies : A Case Study. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the 1^{st} International Workshop on Mobile Agents (MA '97)*, volume 1219 of *LNCS*, pages 39–49, Berlin, Germany, April 1997. Springer-Verlag.

[54] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[55] Li Gong and Xiaolei Qian. Computational issues in secure interoperation. *Software Engineering*, 22(1) :43–52, 1996.

[56] James Gosling, Guy Steele Bill Joy, and Guy L. Steele. *The Java(TM) Language Specification*. Addison Wesley, 1996.

[57] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents : The next generation in distributed computing. In *Proceedings of the Second Aizu International Symposium on Parallel Algorithms and Architectures Synthesis (pAs '97)*, pages 8–24, Fukushima, Japan, March 1997. IEEE Computer Society Press.

[58] Robert S. Gray, David Kotz, George Cybenko, and Daniela Rus. D'Agents : Security in a multiple-language, mobile-agent system. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[59] H. Hosmer. Metapolicies II. In *15th National Computer Security Conference*, pages 369–378, Baltimore, MD, October 1992.

[60] Fabrice Huet. *Objets Mobiles : conception d'un middleware et évaluation de la communication*. PhD thesis, Université de Nice-Sophia Antipolis, 2002.

[61] Mamdouh H. Ibrahim. Reflection and metalevel architectures in object-oriented programming (workshop session). In *Proceedings of the European conference on Object-oriented programming addendum : systems, languages, and applications*, pages 73–80. ACM Press, 1991.

[62] G. Karjoth, D.B. Lange, and M. Oshima. A Security Model for Aglets. *IEEE Internet Computing*, 1(4) :68–77, 1997.

[63] 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.

[64] Neeran M. Karnik and Anand R. Tripathi. Security in the ajanta mobile agent system. *Software, Practice and Experience*, 31(4) :301–329, 2001.

[65] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48 :203–209, 1987.

[66] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC : Keyed-Hashing for Message Authentication*, February 1997. RFC 2104.

[67] Axel W. Krings, Jean-Louis Roch, Samir Jafar, and Sebastien Varrette. A probabilistic approach for task and result certification of large-scale distributed applications in hostile environments. In LNCS Springer-Verlag, editor, *European Grid Conference EGC'2005*, Amsterdam, The Netherlands, february 14-16 2005.

[68] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Conf. on Information Sciences and Systems*, Princeton, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

[69] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10) :613–615, 1973.

[70] J.-C. Laprie et al. *Dependability : Basic Concepts and Terminology*. Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, Wien, New York, 1992.

[71] Arjen K. Lenstra and Eric R. Verheul. Selecting Cryptographic Key Sizes. In *Public Key Cryptography*, pages 446–465, 2000.

[72] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[73] Geoffrey Lewis, Steven Barber, and Ellen Siegel. *Programming with Java IDL*. John Wiley & Sons, 1998.

[74] Mike Lewis and Andrew Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, New York, USA, August 1996. IEEE.

[75] J. Linn. *Generic Security Service Application Program Interface Version 2, Update 1*, January 2000. RFC 2743.

[76] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. SecureUML : A UML-Based Modeling Language for Model-Driven Security. In *UML '02 : Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 426–441, London, UK, 2002. Springer-Verlag.

[77] Sergio Loureiro, Refik Molva, and Yves Roudier. Mobile Code Security. In *Proceedings of ISYPAR 2000*, Toulouse, France, 2000.

[78] Pattie Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, March 1987.

[79] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In T. Helleseth, editor, *Advances in Cryptology — Eurocrypt '93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397, Berlin, 1994. Springer-Verlag.

[80] Andrew J. Maywah. An Implementation of a Secure Web Client Using SPKI/SDSI Certificates. Master's thesis, Massachusetts Institute of Technology, May 1999.

[81] Daryl McCullough. A Hookup Theorem for Multilevel Security. *IEEE Transactions on Software Engineering*, 16(6) :563–568, 1990.

[82] Paul L. McCullough. Transparent forwarding : First steps. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 331–341. ACM Press, 1987.

[83] Gary McGraw and Edward W. Felten. *Securing Java : Getting Down to Business with Mobile Code, 2nd Edition*. Wiley, 1999.

[84] J. McLean. The Algebra of Security. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–7, Oakland, CA, USA, April 1988.

[85] John McLean. A General Theory of Composition for a Class of "Possibilistic" Properties. *IEEE Transactions on Software Engineering*, 22(1) :53–67, 1996.

[86] Ralph C. Merkle. Secure communications over insecure channels. *Communications of ACM*, 21(4) :294–299, 1978.

[87] Sun Microsystems. Java Authentication and Authorization Service (JAAS).

[88] Kevin D. Mitnick and William L. Simon. *The Art Of Intrusion : The Real Stories Behind The Exploits Of Hackers, Intruders, & Deceivers*. Hardcover, 2005.

[89] Richard MonsonHaefel, Bill Burke, and Sacha Labourey. *Enterprise JavaBeans, 4th Edition*. O'Reilly & Associates, 2004.

[90] Alexander Morcos. A Java implementation of Simple Distributed Security Infrastructure. Master's thesis, Massachusetts Institute of Technology, May 1998.

[91] Anand Natrajan, Anh Nguyen-Tuong, and Andrew Grimshaw Marty Humphrey. The Legion Grid Portal. In *Grid Computings - Grid 2001*, volume 2242 of *Lecture Notes in Computer Science*, Denver, Colorado, 2001, November 2001. ACM, IEEE, Springer-Verlag, Heidelberg, Germany.

[92] B. Clifford Neuman. Proxy-based authorization and accounting for distributed systems. In *International Conference on Distributed Computing Systems*, pages 283–291, 1993.

[93] *nQueens n = 25 solved with ObjectWeb ProActive*. http ://www.inria.fr/oasis/ProActive/nqueens25.html.

[94] National Bureau of Standards. Data encryption standard. *Federal Information Processing Standards Publication*, 46, 1977.

[95] Alexandre Oliva and Luiz Eduardo Buzato. Designing a secure and reconfigurable metaobject protocol. Technical Report IC-99-08, University of Campinas, Brésil, February 1999.

[96] Object group management. http ://www.omg.org, 2004.

[97] OSGi Web Site. http ://www.osgi.org, 2005.

[98] Geoffrey A. Pascoe. Encapsulators : A new software paradigm in smalltalk-80. In *Proceedings of OOPSLA'86*, pages 341–346, 1986.

[99] A. Puliafito and O. Tomarchio. Security mechanisms for the map agent system. In *8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000)*, Rhodos, Greece, January 2000.

[100] A. Puliafito, O. Tomarchio, and L. Vita. MAP : Design and Implementation of a Mobile Agents Platform. Technical Report TR-CT-9712, 1997.

[101] Ingo Rammer. *Advanced .NET Remoting, C# Edition*. Apress, 2005.

[102] 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.

[103] 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.

[104] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communication of the ACM*, 21(2) :120–126, 1978.

[105] Aviel D. Rubin and Jr. Daniel E. Geer. Mobile Code Security. *IEEE Internet Computing*, 2(6) :30–34, 1998.

[106] Andrew S, Grimshaw, and William A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1) :39–45, January 1997.

[107] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The nist model for role-based access control : towards a unified standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63, Berlin, Germany, 2000. ACM Press.

[108] Luis F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. In *CCGRID '01 : Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 337, Washington, DC, USA, 2001. IEEE Computer Society.

[109] Andreas Schaad, Jonathan Moffett, and Jeremy Jacob. The Role-Based Access Control System of a European Bank : a Case Study and Discussion. In *SACMAT '01 : Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 3–9, New York, NY, USA, 2001. ACM Press.

[110] Rudiger Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In IEEE, editor, *2001 International Conference on Peer-to-Peer Computing (P2P2001)*, Department of Computer and Information Science Linko pings Universitet, Sweden, august 2001.

[111] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *POPL'84 : conference record of the Annual ACM symposium on Principles of Programming Languages*, pages 23–35, 1984.

[112] 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.

[113] Sun Microsystems. Java remote method invocation specification, October 1998. ftp ://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.

[114] Sun Microsystems. *Enterprise JavaBeans Specification, version 2.0*. Sun Microsystems, August 2001.

[115] J. Tardo and L. Valente. Mobile agent security and Telescript. In *IEEE CompCon*, 1996.

[116] U.S. National Institute of Standards and Technology (NIST). Specification for the Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication 197 (FIPS PUB 197)*, November 2001.

[117] Julien Vayssière. *Une architecture de sécurité pour les applications réflexives, Application à Java*. PhD thesis, OASIS - INRIA Sophia Antipolis, 2002.

[118] Bill Venners. Under the hood : The architecture of aglets. *JavaWorld : IDG's magazine for the Java community*, 2(4), April 1997.

[119] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6) :826–849, 1997.

[120] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In Pierre Cointe, editor, *Proceedings of the second international conference Reflection'99*, number 1616 in Lecture Notes in Computer Science, pages 2 – 21. Springer, July 1999.

[121] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, and F. Siebenlist. X.509 proxy certificates for dynamic delegation. Technical report, 3rd Annual PKI R&D Workshop, 2004.

[122] Von Welch, Frank Siebenlist, Ian Foster, John Bresnahan, Karl Czajkowski, Jarek Gawor, Carl Kesselman, Sam Meder, Laura Pearlman, and Steven Tuecke. Security for grid services. In *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*. IEEE Computer Society Press, 2003.

[123] B. White, A. Grimshaw, and A. Nguyen-Tuong. Grid-based file access : The Legion I/O model. In *9th IEEE Internation Symposium on High Performance Distributed Computing*, pages 165–173, Pittsburgh, Pennsylvania, USA, August 2000. IEEE, IEEE Computer Society.

[124] Brian S. White, Michael Walker, Marty Humphrey, and Andrew Grimshaw. LegionFS : A Secure and Scalable File System Support Cross-Domain High Performance Applications. In *Proceedings of Supercomputing 2001*, Denver, Colorado, USA, November 2001.

[125] Darrel Kienzle William A. Wulf, Chenxi Wang. A New Model of Security for Distributed Systems. In *Proceedings of the UCLA conference on New security paradigms workshops*, pages 34–43, Lake Arrowhead, California, United States, 1996. SIGSAC : ACM Special Interest Group on Security, Audit, and Control, ACM Press, New York, NY, USA.

[126] Philip Zimmermann. Pretty Good Privacy : public key encryption for the masses. In *Building in big brother : the cryptographic policy debate*, pages 93–107, New York, NY, USA, 1995. Springer-Verlag New York, Inc.

Une Architecture de Sécurité Hiérarchique, Adaptable et Dynamique pour la Grille

Résumé

Si la sécurité est une notion essentielle aux applications, particulièrement aux applications distribuées, ses nombreux concepts représentent une étape difficile de leur développement.

Les intergiciels actuels intègrent un grand nombre de technologies relatives aux concepts de sécurité. Cependant, ils laissent aux développeurs la tâche de choisir la technologie la plus adaptée ainsi que la gestion des processus sous-jacents. Cet exercice se révèle d'autant plus difficile lorsque l'application évolue dans un environnement dynamique tel que celui des grilles de calcul.

Afin de faciliter le déploiement d'applications distribuées et sécurisées, cette thèse présente un modèle de sécurité décentralisé permettant aux divers acteurs (administrateurs, fournisseurs de ressources, utilisateur) d'exprimer leurs politiques de sécurité. Son utilisation se veut totalement transparente au code métier des applications. La configuration de la politique de sécurité d'une application est exprimée dans des fichiers externes en dehors du code source de cette dernière. Il est ainsi possible d'adapter la sécurité de l'application en fonction de son déploiement.

Notre mécanisme de sécurité est conçu pour s'adapter dynamiquement aux situations survenant dans le cycle de vie d'une application distribuée, notamment l'acquisition de nouvelles ressources et la création de nouveaux objets.

L'implantation du modèle au sein d'une bibliothèque pour le calcul distribué a permis de démontrer la faisabilité de l'approche et ses avantages. En effet, son implantation transparente a permis son intégration immédiate avec les autres modules de la bibliothèque (communications de groupe, mobilité, composants, pair-à-pair). Les tests de performance réalisés afin d'évaluer son surcoût ont confirmé la validité de l'approche.

Mots-clefs : Sécurité, Grille de calcul, Propagation dynamique, Déploiement, Transparence, Modèle de programmation objet, Protocole à Méta-Objets

---

An Hierarchical, Versatile and Dynamic Security Architecture for The Grid

Abstract

Whereas security is a key notion in the world of distributed applications, its numerous concepts are a difficult step to overcome when constructing such applications.

Current middlewares provide all major security-related technologies. However developers still have to select the more accurate one and handle all its underlying processes which is particurally difficult with dynamic, grid-enabled applications.

To facilitate the use of security concepts in such application, this thesis presents a decentralised security model which takes care of security requirements expressed by all actors (resource providers, administrators, users) involved in a computation.

The model is implemented outside the application source code. Its configuration is read from external policy files allowing the adaptation of the application's security according to its deployment.

It has been conceived to handle specific behaviors which could happen during a distributed application life-cycle (use of newly discovered resources, remote object creation).

Moreover, the implentation within the *ProActive* library has valided the approach and had demonstrated its advantages. Indeed, thanks to its transparency, it has been seamlessly integrated with the other features of the library (migration, group communications, components, peer-to-peer). Benchmarks have consolidated the validity of the approach.

Keywords : Security, Deployement, Transparency, Dynamic propagation, Object Oriented Programming, Meta Object Protocol