UNIVERSITE LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Année académique 2001-2002

Etude critique de mécanismes de sécurité pour l'architecture Jini

Pierre Stadnik

Directeur de Mémoire:
Prof. Esteban Zimányi

Mémoire présenté en vue de l'obtention du grade de licencié en Informatique

Je tiens à remercier toutes les personnes qui ont contribué de près ou de loin à l'élaboration de ce mémoire.

Je voudrais tout d'abord remercier M. Esteban Zimányi pour son accueil et sa gentillesse qui m'ont tout naturellement renforcé dans l'idée de réaliser le mémoire sous sa tutelle.

Mes remerciements vont également à Jean-Michel Dricot pour ses précieuses remarques et pour m'avoir fait découvrir le monde de la technologie Jini.

J'aimerais également remercier tout particulièrement M. Yves Roggeman pour ses précieux conseils sur l'analyse sécuritaire et pour sa disponibilité.

Je tiens aussi à remercier M. Olivier Makowitch pour sa gentillesse et pour ses corrections.

Enfin, j'aimerais remercier ma famille qui m'a supporté durant la dernière ligne droite de ce mémoire et tout particulièrement à Maya qui m'a fortement conseillé et soutenu depuis la naissance de ce travail.

Table des matières

Symboles et conventions

K_P	Clé publique (chiffrement/déchiffrement asymétrique).
K_C	Clé privée ou confidentielle (chiffrement/déchiffrement asymétrique).
K_S	Clé secrète (chiffrement/déchiffrement symétrique).
K_a	Une clé asymétrique quelconque : qu'elle soit privée ou publique.
K	Une clé quelconque : qu'elle soit secrète, privée ou publique.
m	Message clair (suite de bytes) envoyé sur un moyen de transmission.
c	Message chiffré du message « m ».
E(K, m)	Fonction pour chiffrer un message clair « m » à l'aide d'une clé K.
D(K, c)	Fonction pour déchiffrer un message chiffré « c » à l'aide d'une clé K.
$S(K_a, m)$	Fonction de signature pour signer un message clair « m » à l'aide d'une clé asymétrique K_a.
s	Une signature numérique résultant de la fonction $S(K_a, m)$.
H(m)	Fonction de hashage ou de résumé de message « m ».
h	Résumé de message ou hash code produit par H()
MAC	Message Authentication Code : voir fin de section 3.2.4.

DoS	Deny of Service : Type d'attaque visant à rendre un système ou un service inopérant ou sous-efficace. Dans ce mémoire, nous utilisons ce terme pour représenter tout ce qui touche à la disponibilité (des ressources).
SSL	Secure Socket Layer : voir fin de section 3.2.7.
TLS	Transport Layer Security : voir fin de section 3.2.7.
IPSec	Internet Protocol Security : voir fin de section 3.2.7.
ACL	Access Control List : voir section 3.3.
CA	Certificate Authority : voir section 3.2.6.
CRL	Certificate Revocation List : voir section 3.2.6.
PKI	Public Key Infrastructure : Terme générique souvent utilisé pour exprimer « infrastructures pour certificats numériques ».
SPKI	Simple Public Key Infrastructure : voir section 7.2.
X.509	Catégorie d'infrastructure PKI (voir section 3.2.6).
PGP	Pretty Good Privacy : catégorie d'infrastructure PKI (voir section 3.2.6).
Acteur (Jini)	Dans notre analyse sécuritaire de haut niveau, il s'agit de toute entité ayant un rôle particulier dans les communications Jini : C, S, L, P, U, F et B.
B	Serveur de bytecode : peut être un serveur HTTP, FTP ou tout autre protocole permettant le transfert de fichiers. Il fournit les fichiers « class » ou « jar » des proxies aux clients.
C	Client : application Jini exploitant les services S par l'intermédiaire de leur proxy P. Le client appartient à l'utilisateur U.
F	Fournisseur du service S : il est le propriétaire du service S.
L, LUS	Lookup Service : est un service Jini standard dont la vocation est d'enregistrer les proxies P des services S et de les transmettre à ses clients C suite à une demande.

P	Proxy du service S : Il s'agit d'un objet Java implémentant une interface spécifiée par le client et qui est destiné à être exécuté sur la machine du client ($P = P_S + P_B$).
P_S	Proxy State ou Proxy Serialized : il s'agit du proxy vu comme objet sérialisé. Celui-ci reprend la partie « donnée » de l'objet (aussi appelée « état de l'objet ») et une URL indiquant la localisation du serveur de fichier fournissant le bytecode correspondant.
P_B	Proxy Bytecode : il s'agit du code (fichier « class ») du proxy P, le plus souvent stocké dans un fichier « jar » et destiné à être téléchargé à la demande.
S	Service Jini : Application Jini, exécutée sur la machine du serveur et destinée à être appelée par son proxy P qui est exécuté dans la machine du client C.
SL	Secure lookup : voir section 7.1.
U	Utilisateur de C.
…*	P*, C*, S*, L*, B*, U*, F* : tout acteur considéré comme malicieux ou malhonnête. Il cherche à mettre en péril notre système sécuritaire.
JVM	Java Virtual Machine : Il s'agit du logiciel chargé d'interpréter le bytecode. La JVM est le plus souvent chargée au-dessus d'un OS. Elle est l'origine de l'indépendance de plate-forme propre au langage Java.
bytecode	Code machine intermédiaire généré par un compilateur Java et destiné à être exécuté dans une JVM.
jar	Java Archive : fichier compressé au format « zip » destiné à regrouper un ensemble de fichiers « class » Java, des signatures numériques, et d'autres fichiers.
JDK	Java Development Kit : il s'agit d'un environnement de programmation pour le langage Java développé et distribué par Sun Microsystems.
JCE	Java Cryptography Extension, librairie Java qui fournit des primitives cryptographiques.

JSSE	Java Secure Socket Extension : est un package optionnel de Sun Microsystems pour Java qui fournit des sockets SSL ou TLS.
JAAS	Java Authentication and Authorization Service : est une extension de Sun Microsystems qui ajoute des mécanismes pour l'authentification des utilisateurs avec des contrôles d'accès appropriés à la sécurité de l'architecture Java 2.
Marshalling	Processus selon lequel un appel de procédure distante avec ses paramètres sont transformés en un flux de bits pour transmission sur le réseau : voir section 2.4.2.
Marshall Object	Objet Java faisant office de conteneur d'objet Java sérialisé : voir remarque (c) dans la section 5.2.2.
Sérialisé	Objet Java dont l'état (la partie donnée) a été transformé en un flux de bits : voir section 2.3.4.
RPC	Remote Procedure Call : voir section 2.4.2.
RMI	Remote Method Invocation : voir section 2.4.3.
OMG	Object Management Group : organisation regroupant un ensemble de vendeurs de matériel, de systèmes et d'applications. L'OMG produit des spécifications, comme CORBA et UML, notamment pour la promotion de la technologie orientée objet.
CORBA	Common Object Request Broker Architecture : voir section 2.4.4.
IDL	Interface Definition Language : langage de définition d'interfaces génériques spécifié par l'OMG et destiné à l'architecture CORBA.
GIOP	General Inter-Orb Protocol : voir section 2.4.4.
IIOP	Internet Inter-ORB Protocol : voir section 2.4.4.
JINI	« Jini Is Not Initials » ? : ou bien voir chapitre 2.

LAN	Local Area Network : réseau privé d'ordinateurs dont la taille ne dépasse pas quelques kilomètres, les débits sont de l'ordre de 10 à 100 Mbit/s, la topologie est généralement sous forme de bus ou d'anneau, il est souvent un réseau à diffusion et son contrôle est décentralisé.
TTL	Time To Live : Un petit entier se trouvant dans un paquet IP et destiné, par sa décrémentation régulière, à limiter le temps de vie du paquet sur le réseau.
TCP	Transmission Control Protocol : protocole de la couche transport fournissant une connexion fiable sur un réseau IP.
IP	Internet Protocol : protocole de la couche réseau notamment utilisé sur Internet.
UDP	User Datagram Protocol : protocole de la couche transport utilisé pour envoyer des « datagrams » sans garantie d'arrivée sur un réseau IP.
URL	Uniform Ressource Locator, ex. « http://www.ulb.ac.be »
PDA	Personal Digital Assistant : petit ordinateur de poche.
CPU	Central Processing Unit : synonyme de processeur.
OS	Operating System ou système d'exploitation.

Chapitre 1

Introduction

1.1 Contexte du sujet [1]

Les systèmes distribués sont fondamentalement différents des systèmes informatiques isolés ou centralisés. Parmi les caractéristiques les plus fréquemment mentionnées, nous avons : le délai de latence, la bande passante limitée, l'accès mémoire, les défaillances partielles, la concurrence et la consistance (cf. [2]). La sécurité doit sans aucun doute être ajoutée à cette liste car un système distribué requiert la cryptographie alors qu'un système centralisé peut s'en passer.

Plusieurs architectures pour objets distribués tel que CORBA ou Java RMI tentent de masquer, aux yeux du programmeur, les différences entre la programmation localisée et distribuée. Pratiquement, cela signifie que les appels aux méthodes d'objets distants s'effectuent quasiment de la même manière que pour des objets locaux. Ce principe a le gros avantage de rendre la programmation « réseau » beaucoup plus simple à appréhender mais a de nombreux inconvénients. Par exemple, le traitement des défaillances partielles n'est pas pris en compte.

La technologie Jini, développée par Sun Microsystems, prend une approche totalement différente [15]. Jini permet à des entités de se découvrir pour former une communauté de manière spontanée et dynamique, sans intervention humaine. Une entité membre de la communauté est alors en mesure fournir des services aux autres membres du réseau Jini ainsi formé.

[1] Le contexte du sujet étant identique à celui du mémoire [5], nous nous référons à la section 1.1 de ce travail pour réaliser en partie le corps de notre introduction

Comme de tels réseaux sont soumis à de rapides changements de topologie, Jini offre aux développeurs un modèle de programmation qui les force à prendre en compte le fait que le réseau n'est pas fiable. Chaque application nécessite différentes manières de traiter ce problème, et c'est donc pour cela que la considération de cette difficulté permet de développer des applications plus fiables.

Jini est basé sur Java et sur la particularité de ce dernier à pouvoir télécharger dynamiquement du code en provenance du réseau : « le code mobile ». Par ce principe, il offre à ses applications la notion « d'indépendance de protocole » qui le différencie de toutes les infrastructures capables de gérer la découverte de services sur un réseau. Le code mobile permet aussi d'alléger la charge d'exécution généralement attribuée de manière exclusive aux machines qualifiées de « serveur ». De manière générale, Jini semble être une technologie prometteuse pour la découverte des services dans un environnement dynamique et spontané. Cependant, l'architecture Jini ne possède actuellement aucune fonctionnalité sécuritaire autre que celles déjà présentes dans la plate-forme Java 2 sous-jacente comme les méthodes de protection contre l'exécution de code malicieux dans la Java Virtuel Machine (JVM).

1.2 But et plan du travail

Le premier « Jini Starter Kit » de Sun Microsystems est officiellement sorti en 1999. Depuis cet instant, plusieurs travaux se sont intéressés aux problèmes sécuritaires inhérents à son mode de fonctionnement. Ces travaux ont le plus souvent aboutis sur l'implémentation d'une architecture particulière considérée comme idéale pour résoudre le problème de la sécurité. Certaines solutions précisent l'environnement d'application, d'autres restent plus générales. Chacune de ces architectures ayant une vision du problème qui lui est propre, toutes se sont néanmoins focalisées sur les problèmes des communications « réseau » non-sécurisées et sur le problème du « proxy opaque » (cf. section 5.10.4).

Ce mémoire a pour buts principaux de mettre en évidence les problèmes sécuritaires propres à la technologie Jini et de comparer certains travaux déjà réalisés en la matière.

Dans le chapitre 2, nous tentons tout d'abord de définir Jini et son mode de fonctionnement. Pour mieux le situer et motiver son étude, nous effectuons également un travail de comparaison vis-à-vis des autres infrastructures pour objets distribués les plus connues comme RMI et CORBA.

Dans le chapitre 3, nous définissons quelques concepts et techniques de base permettant la mise en œuvre de tout mécanisme sécuritaire. L'intérêt de ce chapitre est de familiariser n'importe quel lecteur avec les notions générales de la sécurité informatique.

Le chapitre 4 expose les principaux outils et atouts sécuritaires de la « plate-forme Java 2 ». Un petit détour sera également effectué du côté du « service sécurité » de l'architecture CORBA.

Le chapitre 5 procède à l'analyse sécuritaire proprement dite de l'architecture Jini. Nous l'observons sous ses aspects bas niveau (fonctionnement technique), haut niveau (interaction entre entités Jini) et aussi sous les aspects sécuritaires le caractérisant par rapport aux autres architectures plus connues. La fin de ce chapitre contient également une liste d'exigences sécuritaires potentielles, avec quelques éléments de réponse et leurs difficultés liées. Cette liste, non exhaustive, mais néanmoins suffisante, nous servira de point de référence pour comparer les différentes architectures proposées dans un chapitre suivant.

Le chapitre 6 tente, quant à lui, d'aborder un aspect plus pratique en donnant une série d'exemples concrets d'utilisation de la technologie Jini dans la vie de tous les jours. Ceux-ci n'ont qu'un but d'illustration, le lecteur chevronné, se fera un plaisir d'appliquer à chacun de ces exemples, l'analyse théorique établie lors du chapitre précédent.

Le chapitre 7 procède à l'analyse et à la critique de deux architectures sécuritaires relativement différentes qui sont proposées pour l'infrastructure Jini : l'une dans le cadre d'un article [7] réalisé par Peer Hasselmeyer, Roger Kehr et Marco Voβ ; l'autre dans le cadre d'un mémoire [5], réalisé par Passi Eronen. Pour ce faire, nous synthétisons l'ensemble des idées particulières ou novatrices de chacun des deux travaux. Un tableau résumé est également accompli afin de donner un rapide aperçu des caractéristiques de l'architecture. Dans un des deux cas, la solution étant moins facile à comprendre, nous résumons également l'architecture proposée. Dans tous les cas nous vérifions l'adéquation de la solution par rapport aux exigences établies à la fin du chapitre 5. A la fin de chacune de ces études, nous discutons les avantages et inconvénients de la solution envisagée.

Le chapitre 8 clôt le travail en établissant les conclusions de notre analyse.

Chapitre 2

Jini et ses concepts de base

Dans ce chapitre, nous tentons de définir Jini. Pour cela, nous commençons par l'observer selon différents points de vue. Ensuite, nous expliquons son histoire et en particulier les motivations de sa création. Nous détaillons également ses mécanismes fonctionnels principaux. Enfin, pour mieux le situer techniquement, nous le comparons à d'autres infrastructures bien connues.

2.1 Jini, qu'est-il exactement ? Explications générales…

Jini est une technologie développée et distribuée par Sun Microsystems. Elle se définit de plusieurs manières suivant l'utilisation que l'on souhaite en faire :

D'un point de vue général, Jini définit les concepts suivants : un réseau Jini appelé généralement « **communauté** » ou anciennement « djinn », est un ensemble d'entités ayant accès les unes aux autres. Deux types d'entités existent dans une communauté Jini. La première est dite « **Service** ». Comme son nom l'indique, cette entité est capable de réaliser une certaine tâche pour l'ensemble des autres entités membres de la communauté. L'autre type d'entité, est le « **Client** ». Celui-ci utilise les services. Remarquons que ce principe n'est cependant pas exclusif dans le sens où un service peut lui-même être un client d'autres services. Ceci nous amène à utiliser le terme générique « service » pour représenter ces deux types d'entités. Chaque communauté regroupe ses membres autour d'un service centralisateur du nom de « lookup ». Son rôle principal est la gestion de la découverte des services se trouvant dans son entourage. Sa participation comme entité de communication intermédiaire entre le

client et le service, se limite au processus de découverte. Une fois établie, la communication s'effectue directement entre le client et le service. Un service n'est pas attaché de manière exclusive à un lookup particulier, il peut être répertorié dans plusieurs lookup en même temps. Le lookup étant un service à part entière, il est également capable de s'enregistrer dans un autre lookup gérant une autre communauté. Nous assistons donc à la possibilité de fédérer des communautés Jini. C'est pour cela que nous parlons également de « **fédérations** » Jini.

D'un point de vue conceptuel, Jini élève la plate-forme Java en lui ajoutant la notion de service. Il permet le développement d'applications distribuées basées sur cette notion. Nous quittons le modèle client-serveur pour passer au modèle client-service. La frontière entre application cliente et application serveur devient plus floue. La machine hébergeant l'application cliente peut maintenant effectuer elle-même, grâce au code mobile, le travail qu'aurait réalisé la machine hébergeant l'application serveur.

D'un point de vue technique, Jini offre une infrastructure logicielle, communément appelée « **middleware** », permettant à des objets Java, appelés services, de se découvrir et de s'utiliser de façon spontanée. L'analogie à « une couche de transport intelligente » est justifiable de ce point de vue. Cette couche middleware peut, vu de l'extérieur, être comparée à l'ORB, le bus logiciel de l'architecture CORBA. Jini exploite cependant la notion de « **code mobile** » fournie par Java et qui permet de transférer un objet Java complet, données et code compris, d'une « Java Virtual Machine » à une autre. Grâce à ce mécanisme, le service fournit dynamiquement au client un objet Java sérialisé capable de « piloter » le service à distance. Cet objet est appelé « **Proxy** ». Le service procure, d'une certaine façon, au client, le code que celui-ci ne possédait pas lors de sa création. Le client sait préalablement quel type de service il veut utiliser et comment l'utiliser mais il ne sait pas comment le proxy et le service réalisent cela. Jini est construit en Java, pour Java et exploite la technologie Java RMI (« Remote Method Invocation » voir section 2.4.3). Chaque entité désirant faire partie de la communauté Jini doit tout au moins posséder une partie de code écrite en Java. Remarquons que seul le client est réellement obligé de posséder une JVM (Java Virtual Machine) afin d'exécuter le proxy qu'il a reçu du lookup.

Du point de vue du développeur, Jini crée un nouveau paradigme de programmation en proposant un modèle de développement d'applications distribuées fiables. Le modèle force le programmeur à prendre en compte les différents problèmes inhérents aux applications réparties (cf. [2]). Ces problèmes sont ceux retrouvés dans toute application séparée par un ou plusieurs réseaux : fiabilité, tolérance aux pannes, défaillance partielle, délai de latence, bande passante limitée, …

D'un point de vue logiciel, Jini est développé en Java et se présente comme une extension logicielle pour la plate-forme Java 2. Il est composé d'un ensemble de classes regroupées dans une collection de fichiers « jar ». Ces classes s'adjoignent aux classes de base de l'environnement Java. Il étend ainsi la hiérarchie des « packages » standards offerts par Java.

D'un point de vue pratique, Jini est une technologie qui a trouvé une certaine vocation pour les équipements électroniques, même si ce n'était pas son but initial. Ceux-ci, dits « **Jini-enabled** », peuvent s'insérer comme service dans une communauté Jini. Nous constatons, de ce point de vue, qu'un service représente aussi bien un équipement matériel qu'une application logicielle. L'adhésion fiable et spontanée du service à la communauté ainsi que la capacité du service à fournir au client le code nécessaire à son utilisation sont les principales motivations de l'utilisation de Jini pour l'interconnexion d'équipements électroniques. Fiable car une déconnexion brutale de l'équipement ne corrompt ni le fonctionnement des applications clientes de l'équipement, ni les autres membres de la communauté. Ils ont été conçus pour prendre en compte ce type d'erreur. Spontanée car la disponibilité ou l'indisponibilité de l'équipement est directement prise en compte par le groupe évitant donc une administration humaine. La notion du « code nécessaire à son utilisation » peut être comparée à celle d'un téléchargement dynamique du pilote logiciel de l'équipement. L'avantage vient du fait que ces pilotes doivent normalement être installés manuellement en fonction de l'OS et de la plate-forme sous-jacente. Or ici, nous sommes dans le monde Java où l'indépendance de plate-forme et le code mobile sont rois.

En résumé, Jini définit un ensemble de conventions. Celles-ci permettent de développer des applications distribuées fiables basées sur la notion de service. Jini spécifie comment les services sont connectés à travers un réseau et comment les services sont fournis aux autres membres de la communauté. Dans l'environnement Jini, les différents membres peuvent apparaître et disparaître de façon spontanée. Les communications entre le client et le service sont laissées au libre choix du service.

2.2 Jini, son histoire

2.2.1 Son passé

Quand les chercheurs de Sun Microsystems eurent l'idée de mettre au point une architecture comme celle de Jini, ce fut à l'origine pour offrir la notion de service au monde Java et exploiter une des caractéristiques de ce langage : « le code mobile ». En effet, l'indépendance de plate-forme offerte par Java engendre la possibilité de

déplacer des objets complets d'un endroit à un autre. L'intérêt était donc à la fois marketing et conceptuel. Marketing, pour promouvoir la plate-forme Java et être les premiers sur le marché à offrir ce genre de technologie. Conceptuel, pour étendre le paradigme de « l'orienté objet » vers le monde des réseaux de services.

2.2.2 Son présent

Très vite, Sun Microsystems se rendit compte que cette architecture pouvait être profitable au marché des équipements électroniques. Ce sera donc la première utilisation que vous rencontrerez, même si, vous ne trouvez sûrement pas, à l'heure actuelle, de produits « Jini-enabled » dans votre supermarché le plus proche. Mais Jini parle avant tout de services et de leur découverte. C'est dans cette optique qu'il peut être utilisé comme architecture de développement d'applications distribuées au sens logiciel du terme.

2.2.3 Son futur

La politique actuelle de Sun Microsystems semble cependant donner à Jini un nouvel envol en le positionnant comme solution pour la gestion de services sur Internet. Cependant Jini est une technologie qui a déjà trois ans et semble en manque de marché. D'autres technologies poussées par Sun comme JAXR semblent prendre le pas sur l'enregistrement et la découverte de services WEB. Peut-être Jini est-il trop en avance sur son temps ? Je pense que le futur de Jini dépend de trois facteurs : une poussée économique par les grands acteurs du marché informatique, une plus grande interopérabilité avec d'autres mécanismes de découverte et enfin l'adjonction de mécanismes de sécurité efficaces !

2.3 Concepts et fonctionnements

2.3.1 Service lookup (lookup Service)

Parmi l'ensemble des services réalisables à l'aide de Jini, le « Lookup Service » (LUS) ou lookup fait entièrement partie des spécifications définies par Sun. Il joue un rôle essentiel dans la communauté Jini. Il est chargé d'enregistrer les services se trouvant à sa proximité, d'entreposer leur proxy, de contrôler la disponibilité de ces services et enfin de répondre aux demandes des clients. Les clients demandent au lookup de rechercher pour eux, un service ayant certaines caractéristiques. Pour exprimer cela, chaque proxy implémente une interface Java précisant le comportement du service, mais ne précisant pas la manière dont celui-ci les réalise.

Nous voyons la distinction classique émise entre le « quoi » et le « comment ». La notion d'héritage d'interfaces en Java est fortement exploitée dans le processus de recherche de services. Nous recherchons des services dont le proxy implémente une interface donnée ou tout au moins une interface appartenant à la descendance de celle recherchée. Par exemple, nous pouvons rechercher les imprimantes du réseau sur base de l'interface « `Printer` ». Nous obtenons dès lors comme résultat les imprimantes laser implémentant l'interface « `LaserPrinter` » qui hérite de « `Printer` » mais également les machines multifonctions qui implémentent à la fois les interfaces « `ColorLaserPrinter` », « `Scanner` » et « `Fax` ». Pour préciser d'avantage la recherche, le proxy est accompagné de paramètres satellites le caractérisant comme : « les attributs du service ». Le lookup lui ajoute également un identifiant numérique unique. Cette notion d'attribut (du service) n'est pas à confondre avec les attributs d'une classe (la partie donnée) dans le cas d'un langage orienté objet.

2.3.2 Protocole de découverte (Discovery Protocol)

Une des spécificités importantes de Jini, comme nous le verrons plus tard, est son indépendance de protocole de communication utilisé entre le proxy et le service qu'il représente. Mais Jini définit quand même un protocole nécessaire à son bon fonctionnement. Il s'agit du protocole de découverte utilisé entre le service lookup et ses clients (qui sont les clients d'autres services et ces services eux-mêmes). Dans la spécification actuelle, ce protocole est décrit pour fonctionner au-dessus d'un réseau TCP/IP. Pour être plus précis, ce protocole de découverte en comprend trois praticables :

- Le « **unicast discovery protocol** » est utilisé quand un client connaît le port du service lookup qu'il cherche à joindre et l'adresse IP de la machine serveur sur laquelle se trouve ce lookup (connu). L'objet proxy représentant le service lookup est alors transféré vers le client par l'intermédiaire d'une connexion TCP.

- Le « **multicast discovery protocol** » est utilisé lorsqu'un client désire connaître tous les services lookup (inconnus) se trouvant dans ses alentours. Pour ce faire, le client envoie un paquet multicast UDP avec un petit time-to-live (TTL) sur un port pré-défini. De cette manière nous ne touchons qu'un environnement restreint. Tous les services lookup recevant ce paquet, répondent en utilisant une version serveur de l'unicast discovery protocol.

- Le « **multicast announcement protocol** » est utilisé par le service lookup pour annoncer périodiquement sa présence aux clients. De cette façon, les services sont directement mis au courant de son indisponibilité, suite à une panne matérielle ou à une erreur logicielle par exemple. De même, lorsqu'il revient en ligne, les services peuvent à nouveau s'enregistrer auprès de lui s'ils le désirent. Chaque annonce de présence contient l'adresse IP et le port du lookup de manière à ce que les clients intéressés, c'est à dire les autres entités Jini, puissent lui répondre à l'aide de l'unicast discovery protocol.

De manière générale, lorsqu'un client recherche un lookup connu ou inconnu, il envoie un petit paquet de découverte UDP/IP (voir Figure 2.1) contenant les informations suivantes : la version du protocole de découverte utilisé, le port TCP sur lequel le lookup peut « rappeler » le client, le groupe dans lequel doit appartenir le lookup visé, un ensemble de noms de groupes de services que le lookup doit gérer et les numéros de lookup non concernés par la requête. Le client doit donc ouvrir une connexion TCP/IP et être en attente d'une réponse transmise sur ce port, par un lookup concerné.

Le lookup ayant reçu le paquet UDP/IP de découverte du client, lui transmet alors son proxy à l'aide d'une connexion TCP/IP. Il est très important de remarquer que ce transfert n'est absolument pas basé sur RMI. A la place, le lookup utilise la capacité de sérialisation d'un objet qui fait partie des facilités du langage de programmation Java (voir section 2.3.4). Ce n'est que par après que ce proxy exploitera éventuellement RMI pour ses communications avec le lookup.

Nombre	Type Sérialisé	Description
1	`int`	version du protocole
1	`int`	port TCP où se connecter
1	`int`	nombre de lookups entendus
variable	`net.jini.core.lookup.ServiceID`	les lookups déjà entendus
1	`int`	nombre de groupes
variable	`java.lang.String`	les groupes

Figure 2.1 : Contenu d'un paquet de découverte

Remarque : Nous voyons que la topologie du réseau Jini est intrinsèquement dépendante de la topologie du réseau physique sous-jacent défini entre autres par le protocole TCP/IP.

2.3.3 Processus d'adhésion (Join Process)

Lorsqu'un service reçoit le proxy du lookup sur lequel il désire être enregistré, il le fait grâce à la méthode (`register`) de l'interface (*`ServiceRegistrar`*). Cette interface est standardisée pour tous les proxies de lookups Jini. De cette manière le service client a la capacité de transmettre au lookup son propre proxy. Remarquons qu'à nouveau, le proxy contacte son service, ici le lookup, par le moyen de son choix. Cela peut être via RMI, via des sockets, via un protocole série ou tout autre protocole propriétaire. Ceci dit, l'implémentation fournie par Sun Microsystems exploite RMI pour les communications avec le LUS.

2.3.4 Le Proxy

Le proxy ainsi transmis, est un objet Java sérialisé. Un objet Java sérialisé est une instance d'une classe Java que nous avons transformée en un flux de bits pour son éventuel transfert d'une machine à une autre ou pour son stockage sur un média physique persistant (comme une disque dur, par exemple). Seules les informations caractérisant l'objet, en tant qu'instance de classe, sont sauvegardées. Le bytecode quant à lui réside dans le fichier « class » correspondant à l'objet désigné. Un objet Java sérialisé comportera donc deux éléments. Le premier est l'état de l'objet comme les valeurs de ses attributs : c'est donc la partie « donnée » d'un objet. Le deuxième est son « **codebase** » qui est une URL pointant sur un serveur de fichier capable de transférer, à la demande, le bytecode de l'objet en question (fichier « class ») si ce code ne se trouve pas déjà sur la machine du destinataire. Nous constatons donc que le proxy ainsi transféré n'est pas accompagné du code de ses méthodes. Ceci est dû à la sémantique de RMI qui est utilisé en Jini pour le transfert des objets. Remarquons que le protocole de transfert n'est pas figé. Java permet d'exploiter les protocoles « HTTP », « FTP » de même que n'importe quel autre protocole personnel. L'URL, créée dans ce dernier cas, peut par exemple avoir la forme suivante : « monprotocole://monserveur:8080/ … ». Lors de sa transmission du service au lookup, le proxy est complété par un ensemble d'éléments satellites le caractérisant : « les attributs du service ». Le lookup lui fournira par la suite un identifiant numérique unique. Ces informations sont destinées au lookup et permettent aux clients de faciliter leur recherche de services. Le proxy s'exécute dans la JVM de l'application cliente, donc en local par rapport à cette dernière. Il peut être une simple souche RMI qui traduit les appels de méthodes du monde objet vers un flux de communications réseau ou bien, il peut être plus évolué et implémenter une part ou la totalité des fonctionnalités du service qu'il représente. Dans ce dernier cas, aucune communication réseau n'est nécessaire entre le proxy et son service. Nous constatons que dans la philosophie Jini, le service offert à proprement parler correspond au couple « proxy-service ».

2.3.5 Processus de recherche (lookup Process)

La recherche de services se déroule plus ou moins de la même manière que l'enregistrement du service dans le sens ou il s'agit d'utiliser une méthode (`lookup`) sur le proxy du lookup Service que nous avons reçu préalablement (voir Discovery Process). La différence ici vient du fait que nous sommes un futur client de service. Nous demandons au LUS de nous fournir les proxies des services qui correspondent à une certaine interface ainsi qu'aux attributs énoncés.

2.3.6 L'indépendance de protocole

L'indépendance de la technique de communication mise au point entre le proxy et son service ainsi que sa capacité d'implémenter une part de l'intelligence du service du côté du client, donne à Jini une flexibilité qui le distingue de toutes les autres technologies visant à gérer des objets distribués dans un environnement évolutif spontané. Cette indépendance est donc un des principaux atouts de cette technologie et il s'agit de la préserver. Nous verrons plus loin que celle-ci pose de sérieux problèmes sécuritaires…

2.3.7 Réservations (Leases)

La notion de fiabilité a déjà été soulignée un peu plus haut. En effet, Jini doit être capable de traiter les inévitables problèmes comme les pannes de réseau, les pannes de machine et les erreurs de programmation. Mais comment Jini réalise-t-il cette prouesse ? Principalement par un mécanisme d'**auto-réparation**. Les services et, en particulier, le lookup peuvent exploiter le principe de réservation de ressources. Chaque service peut être vu comme une ressource pour les clients qui l'exploitent. Pour ce faire, chaque client reçoit, suite à une première proposition (négociée) faite par lui, une sorte de ticket de réservation pour une période donnée. Si au bout de cette période, le client n'a pas renouvelé explicitement son intérêt pour le service, celui-ci considère que le client n'est plus là. La réservation peut cependant être interrompue prématurément par l'une des deux parties, mais c'est toujours le service qui a le pouvoir de décision final puisque c'est lui qui offre ses « services ». Comme vous le constatez, la notion de service est une notion générique, le mécanisme de réservation s'étend aussi bien vers les services traditionnels que vers le lookup qui est également un service à part entière. Le lookup a cependant la particularité d'exploiter intensivement ce procédé. Dans ce cas là, le service (client du lookup) propose la période de réservation au lookup par l'intermédiaire de la méthode « `register` » de l'interface « *`ServiceRegistrar`* » implémentée par le proxy. En échange, le service reçoit un objet « `ServiceRegistration` » auquel il peut demander le bail réellement accepté par le lookup.

2.3.8 Les groupes

Jini intègre une notion de groupe. Celle-ci est très limitée et ne correspond pas à un mécanisme de sécurité élaboré. Cette notion permet d'affiner les recherches de services et entraîne donc un meilleur classement de ceux-ci. Chaque client du lookup précise en s'enregistrant dans quel(s) groupe(s) il désire être répertorié. Chaque paquet réseau émis lors du processus de découverte comprend également le ou les groupe(s) visé(s). La notion de groupe permet donc également de limiter le processus de découverte : seuls les lookups satisfaisant un des groupes demandés par le client répondra « présent » à l'appel. Parmi la multitude des groupes définissables, il y en existe un qui est proposé par défaut : le groupe « public ».

2.3.9 Autres notions

Jini apporte en plus du lookup et du mécanisme de réservation, les événements distribués : « **Distributed Events** ». Ceux-ci sont similaires aux événements Java générés sur une machine isolée mais sont portés au niveau des réseaux de services. Jini définit aussi la notion de « **Transaction** » qui permet à des applications distribuées travaillant en coopération de ne pas se retrouver ensemble dans un état incohérent. Le « Transaction Manager » est là pour veiller à l'atomicité des actions. Jini propose également un modèle d'espace de stockage d'objets distribués appelé « **JavaSpaces** ». D'autres services font également partie des spécifications Jini comme « **l'Event Mailbox** » et les « **Lease Renewal Services** ».

2.3.10 Jini : Schéma résumé de son modèle conceptuel

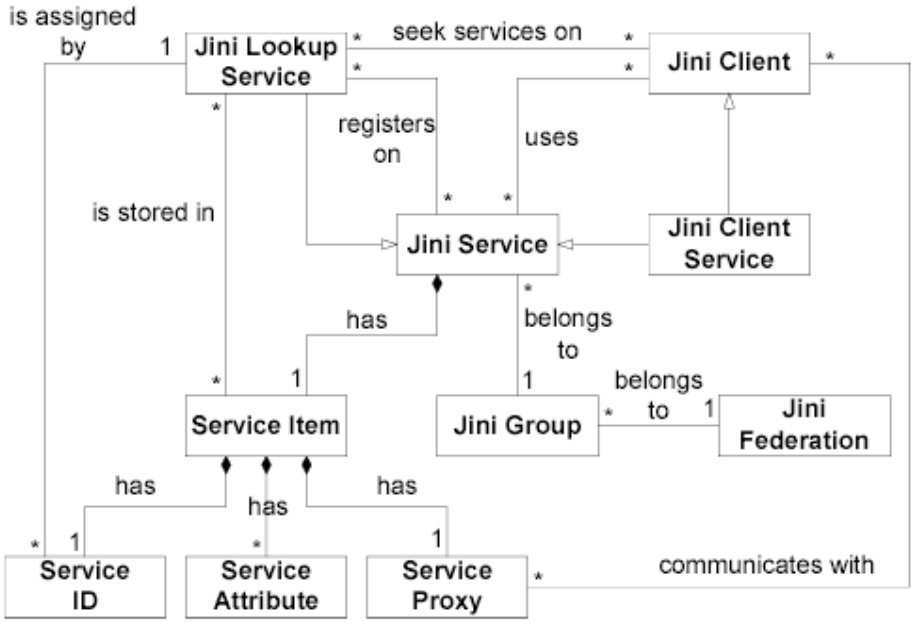

Figure 2.2 : Modèle conceptuel de Jini

Remarque : La notation utilisée est celle conforme aux diagrammes de classes U.M.L.

2.4 Comparaisons techniques

Dans cette section, nous situons et comparons les différentes technologies qui sont souvent confrontées à Jini. Nous procédons, dans une première phase, à la définition de ces technologies afin d'être en mesure de les comparer à Jini durant la deuxième phase.

2.4.1 Protocole de découverte de services

Comme nous l'avons déjà précisé, la technologie Jini regroupe plusieurs notions. Parmi celles-ci, celle du protocole de découverte est souvent utilisée, à tort, pour définir exclusivement Jini. C'est dans cette optique, que nous le trouvons souvent comparé à d'autres protocoles de découverte comme « U-PnP », « Salutation », « Service Location Protocol », et bien d'autres… Jini étant plus qu'un simple protocole de découverte et l'intérêt de ce mémoire portant d'avantage sur ses implications en sécurité au niveau logiciel, une comparaison technique à ce niveau sort du cadre de ce travail.

2.4.2 RPC (Remote Procedure Call)

Le RPC (Remote Procedure Call) est la toute première technique permettant l'abstraction du réseau aux yeux du programmeur. En effet, celle-ci permet à des programmes écrits dans un langage procédural d'appeler, de façon transparente, une fonction se trouvant sur une autre machine séparée par un réseau. La sémantique de « l'appel de fonction » traditionnel est préservée : lorsque l'application appelle une fonction ou une procédure distante, elle se retrouve dans un état « bloquant » jusqu'à la réponse du serveur. Pendant ce temps, l'exécution à proprement parler de la fonction, s'effectue sur la machine distante. Nous sommes donc en présence d'un mécanisme « client - serveur » levé au niveau des appels de fonctions. Pour arriver à cette transparence, chaque application cliente doit posséder, pour chacune des procédures distantes qu'elle compte utiliser, un petit élément logiciel appelé « souche » ou « **stub** ». Celui-ci est chargé de la transformation (« **marshalling** ») des appels de fonction et de leurs paramètres en un flux de bits capable d'être transmis sur le réseau. Une « souche » est d'une certaine façon un substitut local de la procédure distante. De même, chaque application serveur doit posséder le symétrique appelé « squelette » ou « **skeleton** ». Celui-ci est chargé d'effectuer l'opération inverse qui consiste à restaurer (« **unmarshalling** ») le flux de bits en un appel de procédure classique avec ses paramètres. Les souches et les squelettes forment donc une couche logicielle intermédiaire de communication entre les fonctions et le réseau. Pour chaque appel de fonction, le serveur génère en général un nouveau processus ou « thread » de façon à pouvoir répondre à plusieurs clients de manière simultanée. D'autres techniques comme les mémoires partagées ou les files d'attente sont également envisageables. Les types de données échangés par l'intermédiaire des paramètres des fonctions sont simples (entiers, flottants, bytes, …). Nous ne parlons pas d'objet car nous sommes dans la programmation procédurale. Cependant, il est possible d'échanger des structures plus complexes résultant de l'agrégation de plusieurs types simples. Il est clair que la représentation d'une même information peut varier d'un système informatique à un autre si ceux-ci ne sont pas basés sur la même plate-forme logicielle et/ou matérielle. L'interopérabilité entre langages et systèmes différents dépend donc fortement de l'utilisation d'un langage commun de représentation des données échangées sur le réseau.

2.4.3 RMI (Remote Method Invocation)

RMI est l'infrastructure développée par Sun Microsystems pour étendre la programmation orientée objet de son langage Java au monde des réseaux. Il peut donc tout naturellement être classé parmi les infrastructures pour objets distribués. RMI fait de plus intégralement partie du SDK distribué par Sun Microsystems. D'un point de

vue fonctionnel, il s'apparente au RPC appliqué aux méthodes des objets Java. Mais il est accompagné d'un ensemble de caractéristiques supplémentaires.

Comme pour le RPC, chaque objet distant nécessite l'utilisation d'un « stub » et d'un « skeleton » pour le bon fonctionnement des communications (voir Figure 2.3). La terminologie exploitée dans la section précédente reste donc d'actualité pour le RMI. Ces « stub » et « skeleton » sont des objets Java générés automatiquement par un compilateur appelé « rmic ». Celui-ci effectue sa tâche sur base de l'interface que doit implémenter tout objet Java désirant être utilisé comme objet distant (c'est à dire, distant depuis une autre JVM). Les objets distants dans la terminologie RMI sont appelés « Remote Objects ». Ils sont censés implémenter une interface Java qui hérite elle-même de l'interface prédéfinie « `java.rmi.Remote` ». Le protocole de communication établi entre le client et l'objet distant fait partie des spécifications RMI et n'est donc pas « indépendant » (au sens déjà évoqué pour Jini).

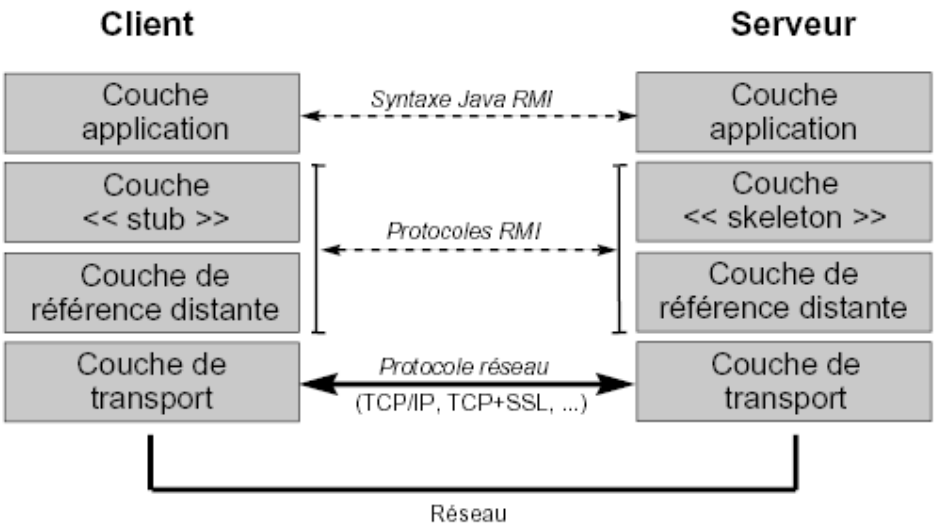

Figure 2.3 : Couches de l'architecture RMI

RMI, comme Jini, possède une caractéristique qui le distingue de toutes les infrastructures pour objets distribués. Il s'agit de sa possibilité de télécharger dynamiquement le « stub » d'un objet distant utilisé par une application Java cliente si ce « stub » n'est pas déjà présent sur la machine du client. Le principe s'étend plus exactement à n'importe quel objet Java sérialisé transmis d'une JVM à une autre par l'intermédiaire des paramètres des méthodes d'objets distants (cf. notion d'objet sérialisé dans la section 2.3.4).

RMI comprend également un mécanisme pour localiser les objets remote. Un objet remote doit normalement s'inscrire dans un petit serveur d'enregistrement appelé « **RMIregistry** » qui associe la référence de cet objet (sa localisation) à un nom

unique au niveau de ce serveur. Le RMIregistry n'est pas un objet Java et est donc dépendant de la plate-forme sous-jacente (en l'occurrence, l'OS). Le développeur d'une application cliente RMI doit connaître préalablement l'interface Java de l'objet distant, l'adresse réseau du serveur RMIregistry et le nom qui est associé à l'objet remote dans ce serveur. Ce sont les trois contraintes obligatoires pour développer un client RMI. Le mécanisme de téléchargement dynamique de code nous évite de posséder préalablement le code du « stub » des objets distants utilisés.

RMI intègre également d'autres notions supplémentaires comme :

- La « **RemoteException** » : qui est une exception générée par le « stub » d'un objet distant et que doit prendre en compte toute application cliente exploitant un objet remote. Cette exception permet d'informer le client d'une mauvaise connexion réseau, par exemple.

- Le « **Distributed Garbage Collector** » : qui est une extension au niveau du réseau du « Garbage Collector » de Java.

- Les « **Activable Objects** » : qui sont des objets persistants qui peuvent être réveillés (activés à la demande par un client (voir « rmid »).

- Les « **Sockets Factories** » : qui permettent de changer le protocole de transport sous-jacent à RMI (TCP/IP, TCP+SSL,…).

2.4.4 CORBA (Common Object Request Broker Architecture

CORBA est une architecture logicielle réalisée par l'OMG (Object Management Group) dont le but est de spécifier et de standardiser un environnement permettant à des **objets hétérogènes** et **distribués** de communiquer entre eux. Ces objets sont dits distribués car nous pouvons y accéder à distance au travers d'un réseau. Ils sont hétérogènes car ils peuvent être issus de langages et de systèmes différents.

L'**ORB** (Object Request Broker), sorte de « bus logiciel », est l'infrastructure nécessaire à cette communication. Celle-ci est installée sur chaque poste désirant accueillir des objets CORBA. Elle joue le rôle d'intermédiaire entre chaque objet et s'occupe donc du contact (la localisation des objets dans l'environnement et de la transmission des informations échangées entre les objets CORBA). Nous retrouvons à nouveau le principe de « stub » et de « skeleton » évoqué pour le RPC et RMI. Ceux-ci viennent se placer entre les applications et l'ORB (voir Figure 2.4). Remarquons que contrairement à RMI, un client doit impérativement posséder le « stub » de l'objet CORBA distant qu'il souhaite appeler.

Pour les échanges d'informations entre objets CORBA, l'OMG a défini un protocole standard de communication (**GIOP** : General Inter-Orb Protocol) dont **IIOP** (Internet IOP) est l'implantation au-dessus de TCP/IP. Remarquons que GIOP nécessite d'être implanté au-dessus d'un protocole de transport fiable, orienté connexion. Il existe également des ponts permettant l'interopérabilité de ce protocole avec d'autres « middleware » : RMI/IIOP, CORBA/COM, …

Chaque objet CORBA est référencé à l'aide d'une sorte de pointeur générique (**IOR** : Interoperable Object Reference) précisant le type de l'objet, sa localisation sur le réseau et son identité (clé) sur la machine « serveur » désignée. Ce pointeur permet donc d'identifier de manière unique un objet CORBA. Remarquons que cet identifiant est plus ou moins transparent pour le programmeur qui manipule les objets CORBA dans son langage de programmation orienté objet grâce à une classe définie par l'OMG. Par exemple, en Java, nous avons la classe « `org.omg.CORBA` ».

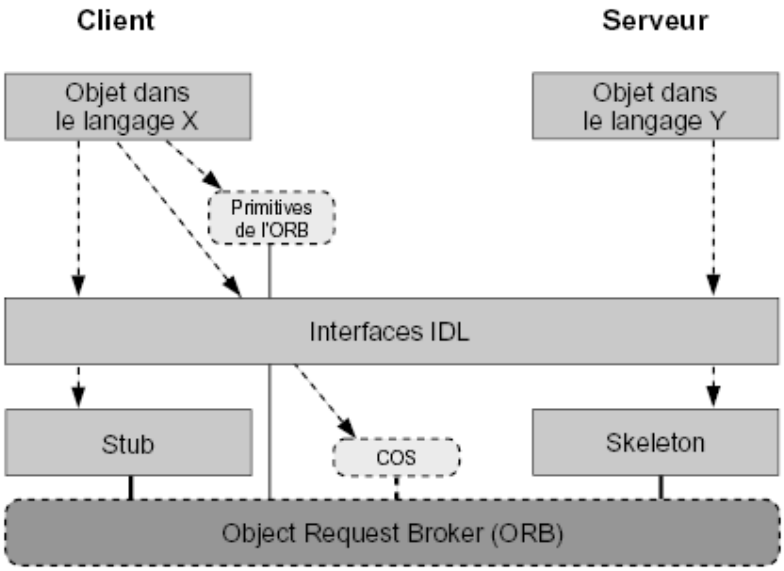

Figure 2.4 : Vue générale de l'architecture CORBA

Pour permettre l'interopérabilité entre objets hétérogènes, l'OMG a défini un langage général de définition d'interfaces (**IDL** : Interface Definition Language) permettant de spécifier de manière exhaustive les attributs et les méthodes que chaque objet CORBA offre aux autres. Nous retrouvons à nouveau le mécanisme « client-serveur » évoqué pour le RPC, à la différence près que nous parlons ici d'objets et de leurs méthodes (comme pour RMI). Les appels sont donc également « bloquants » et l'exécution du code de la méthode se fait sur la machine hébergeant l'objet appelé. Remarquons qu'il existe cependant la possibilité de spécifier, dans la définition de l'interface IDL de l'objet serveur, des méthodes dites « oneway » : Ceci signifie que l'objet client ne devra pas attendre un retour de résultats de la part de l'objet serveur

et qu'il ne sera donc pas dans un état « bloquant ». Il n'y a en fait aucun retour de résultat dans ce cas là. Le langage IDL permet donc de définir les paramètres des méthodes des objets CORBA. Ces paramètres peuvent comme pour le RPC être de type simple ou structuré. Mais ils peuvent également être des objets CORBA. Cependant, nous ne pouvons pas parler de « transfert d'objet » car dans ce cas là, il s'agit du transfert de la référence vers l'objet distant (IOR). Remarquons également que CORBA effectue un « down-casting » des classes des objets passés en paramètre des méthodes : il tronque la classe des objets transmis vers la classe précisée par le paramètre. Ainsi, si nous désirons transmettre un objet de classe « Chien » par un paramètre de méthode étant de la classe « Animal », l'objet reçu au bout du compte sera un « Animal » car le destinataire n'est pas forcément censé posséder le « stub » relatif aux objets de la classe « Chien ».

CORBA est souvent considéré comme un « dinosaure » car ses objectifs sont ambitieux. Un point important caractérisant CORBA, notamment par rapport à RMI et Jini, est qu'il comprend également la spécification d'un ensemble impressionnant de services standards sous forme de classes de « services systèmes » de base : les **COS** (Common Object Services). Ils comprennent par exemple les services : « **Naming Service**», « Persistence Service », « **Trading Service** », « Event Service », « Concurrency Service », « Transaction Service », « **Security Service** »,… Ceux-ci sont accessibles directement à travers l'API de l'ORB ou via d'autres services de recherche d'objets comme le « Naming Service ». Ce dernier peut être comparé au « RMIregistry » car sa fonction est de localiser et fournir une référence vers un objet distant sur base de son nom. Mais le « Naming Service » de CORBA est un objet CORBA à part entière !

D'autres facilités logicielles pour les traitements courants (Common Facilities) ou pour des domaines d'application spécifiques (Domain Interfaces) sont également intégrées à l'architecture.

Dans le vocabulaire de CORBA, chaque objet serveur est considéré comme un objet fournissant des services (via ses méthodes) et non comme étant un service à part entière. Ce qui est l'inverse de Jini. Comme les objets sont hétérogènes, il n'y a pas la notion de code mobile : chaque objet CORBA peut se trouver sur une plate-forme différente ce qui signifie que le code compilé pour la plate-forme de départ ne sera pas forcément exécutable sur la plate-forme d'arrivée. Le code mobile n'a pas de sens dans le monde CORBA. Ceci renforce le fait qu'un « objet », normalement caractérisé par ses données et le code de ses méthodes, ne peut en théorie pas se mouvoir d'une machine à une autre dans un système CORBA.

CORBA est une très belle architecture au sens théorique et il voit son standard évoluer fortement. Il existe de nombreuses implémentations offrant, chacune, des fonctionnalités et des performances différentes. Mais CORBA reste compatible avec son passé et souffre ainsi d'une significative lourdeur au niveau des ressources qu'il exige de la part des systèmes.

2.4.5 Comparaisons par rapport à Jini

RPC par rapport à Jini

Comparer RPC à Jini n'a pas beaucoup de sens. RPC a plus de points communs avec RMI ou CORBA. Si nous l'avons étudié dans une des sections précédentes (section 2.4.2), c'est pour expliquer les bases sur lesquels RMI et CORBA sont bâtis au niveau fonctionnel.

RMI par rapport à Jini

La différence principale entre RMI et Jini vient du fait que Jini est bâti sur RMI. Il hérite donc de l'ensemble des caractéristiques de celui-ci. Ce qui n'est pas la situation inverse. Il est donc plus facile d'énumérer ce que Jini apporte en plus de RMI.

- Jini exploite intensivement le mécanisme de téléchargement dynamique du code d'un objet pour fournir le proxy du service au client à travers le ou les lookup.

- Jini ajoute le protocole de découverte. Celui-ci permet la découverte et l'acquisition du proxy des services lookup qui, à leur tour, vont nous aider à découvrir des objets Java (autres services) non plus sur base de leur nom (comme en RMI) mais plutôt sur base de leur « apparence » (l'interface du proxy et les attributs du service).

- Grâce à ce procédé, un développeur ne doit plus connaître au préalable l'emplacement du RMIregistry ainsi que le nom de l'objet recherché. Tout ce dont il a besoin, c'est d'une interface Java utilisée pour la recherche.

- Un proxy Jini peut être un « stub RMI » mais pas l'inverse. Un proxy Jini offre beaucoup plus de flexibilité. Un « stub RMI » a pour rôle de traduire et de mettre en forme les appels de méthodes de l'interface qu'il implémente vers un flux de bits. Le protocole de communication qu'il emploie avec le « skeleton » du serveur est standard, c'est à dire fixé par RMI. Le proxy Jini, quant à lui, n'est pas limité à un protocole de communication particulier et peut même exécuter sur la machine du client une partie ou la totalité du travail effectué par le service.

- Jini ne possède pas, dans ses spécifications, de « Naming Service » ayant la même fonction qu'un RMIregistry. Mais rien n'empêche un développeur de créer un service Jini remplissant ce rôle ou bien d'exploiter le RMIregistry lui-même.

CORBA par rapport à Jini et RMI

Au niveau fonctionnel, CORBA est probablement plus proche de RMI et de RPC. Nous énumérons ici les différences essentielles entre CORBA, Jini et RMI.

- CORBA utilise le protocole GIOP pour la communication entre les objets. Celui-ci est fixé d'avance et est l'une des clés de l'aspect hétérogène de CORBA. Jini pour sa part prône l'indépendance de protocole entre l'objet proxy et l'objet service. De ce point de vue, rien n'empêche un proxy Jini d'implémenter le protocole GIOP pour communiquer avec un système CORBA.

- CORBA possède toute une panoplie de services prédéfinis que ne possède pas (actuellement) Jini.

- Les services CORBA sont exclusivement exécutés sur la machine hébergeant le service en question (la machine serveur). Jini pour sa part permet d'exécuter une partie ou la totalité du service sur la machine du client par l'intermédiaire du proxy.

- Le service « Trader » de CORBA est comparable au service « lookup » de Jini. Cependant, ce dernier offre plus de flexibilité au niveau de la recherche : celle-ci peut être affinée par des « attributs de service » et par l'identifiant unique du service. Mais la grande différence entre ces deux services résulte de ce qu'ils renvoient au client. Dans le cas de CORBA, il s'agit d'une référence (IOR) vers l'objet CORBA recherché. Dans le cas de Jini, il s'agit de l'objet proxy du service.

- Au niveau de la stabilité de la plate-forme, CORBA ne précise pas le comportement du système lorsqu'une requête vers le service Trader échoue. Jini de son côté, génère des exceptions Java récupérables par le développeur. Celui-ci est donc à même de prendre les mesures qu'il considère nécessaires.

- Tout client d'un objet CORBA doit obligatoirement posséder le « stub » de celui-ci ! Sans cela, il n'y a pas de communication réalisable. Cette exigence n'existe pas pour Jini et RMI car ils jouissent tous les deux du code mobile.

- Un « stub » d'un objet CORBA ne dialogue normalement qu'avec le « skeleton » de l'objet CORBA pour lequel il a été généré. Le proxy de Jini peut pour sa part dialoguer avec tous les services qu'il désire. Il n'est pas limité au simple dialogue

avec son service d'origine. Le proxy, contrairement au « stub », n'est pas généré mais programmé par le développeur du service.

- CORBA est hétérogène. Jini et RMI ne le sont pas car ils sont intrinsèquement liés à Java : ils n'existent pas sans lui.

- CORBA ne peut pas prétendre déplacer des « objets » d'une machine à une autre car, de par son hétérogénéité de plate-formes et de langages, il ne peut offrir le code mobile. CORBA est par contre en mesure de transférer des structures de données ou des références vers d'autres objets CORBA.

- CORBA effectue un « down-casting » des objets lors de leur passage dans un paramètre d'une méthode d'un autre objet CORBA (voir section 2.4.4). Le paradigme de « l'orienté objet » n'est donc pas totalement atteint dans l'architecture CORBA. Jini et RMI, pour leur part, sont conformes à ce paradigme.

- Grâce au principe du « Lease » applicable aux services et en particulier au « lookup Service », Jini offre un modèle de développement d'applications distribuées fiables et tolérantes vis à vis des problèmes inhérents aux réseaux. La spécification des objets CORBA ne prévoit pas un tel mécanisme.

2.4.6 Tableau récapitulatif des comparaisons

	RPC	**RMI**	**CORBA**	**JINI**
Complexité architecture	très légère	légère	lourde	légère
Complexité apprentissage	très facile	facile	difficile	facile
Code mobile	non	oui	non	oui
Indépendance du langage	(oui/non) [6]	non	oui	non
Indépendance OS	(oui/non) [6]	oui	oui	oui
Indépendance µP	(oui/non) [6]	oui	oui	oui
Possibilité d'interagir avec d'autres langages	(oui/non) [6]	(oui/non) [7]	oui	oui
Service « de courtage »	non	non	*Trader Service*	*lookup Service*
Service « de nommage »	non	*RMI Registry*	*Naming Service*	non
Orienté objet	non	oui	oui	oui

	RPC	RMI	CORBA	JINI
Passage de types « simples » comme paramètres	oui	oui	oui	oui
Passage d'objets comme paramètres	non	oui	(oui/non)[1]	oui
Passage d'objets par valeur	non	oui	(oui/non)[2]	oui
Passage d'objets par référence	non	oui	oui	oui
Passage et utilisation d'objets dérivés (donc potentiellement inconnus)	non	oui	non	oui
Indépendance du protocole de communication réseau entité cliente – entité distante(4)	non	(oui/non)[5]	non	oui
Possibilité d'exécuter le service chez le client	non	(oui/non)[3]	non	oui

(1) Dans le cas de CORBA, les objets sont définis au niveau de l'interface, grâce au langage IDL, par l'intermédiaire de types structurés. La partie de l'objet qui peut donc être transférée à travers les paramètres de la méthode est celle correspondant aux données. Les méthodes de l'objet transféré, donc le code, ne peuvent pas être copiées car l'objet client n'utilise pas forcément la même plate-forme.

(2) Depuis la version de CORBA/IIOP 2.2, le support du passage d'objets par valeur a été ajouté, mais cette fonctionnalité n'est pas supportée par toutes les implémentations de CORBA et est soumise à certaines contraintes à cause de l'aspect « hétérogène » de CORBA.

(3) Ce n'est pas prévu comme mécanisme de base de l'architecture RMI. Mais par-dessus, un objet distant RMI peut très bien renvoyer en paramètre de retour à un objet client, un objet que ce dernier activera dans sa machine virtuelle Java.

(4) Autrement dit, le client peut-il communiquer avec l'entité distante d'une autre façon qu'à travers un ORB ou un mécanisme pré-défini ?

(5) Java RMI offre la possibilité de se créer des « Socket Factory ». Ceci permet de redéfinir le protocole de transport entre le client et le serveur et de passer

par exemple par SSL pour l'authentification et pour le chiffrement de l'information transmise par les « sockets ».

(6) L'indépendance dépend essentiellement de l'utilisation d'un langage commun pour la représentation des données transmises sur le réseau. Par exemple, chaque système ne représente pas ses entiers de la même manière.

(7) RMI est prévu pour communiquer entre programmes tournants en Java dans une JVM. Il existe cependant des efforts visant à augmenter l'interopérabilité entre CORBA et RMI, notamment grâce à « RMI over IIOP ».

Chapitre 3

Notions de sécurité informatique

Dans un premier temps, il s'agit de se mettre d'accord sur ce que représente la sécurité informatique. Un petit rappel de ses concepts de base est donc indispensable. Enfin, nous introduisons et nous expliquons rapidement quelques techniques actuelles, comme la cryptographie. Nous pouvons ainsi montrer qu'elles satisfont bien toutes les conditions de notre définition de la sécurité informatique.

3.1 Définitions générales

Suivant Dieter Gollmann [1], la sécurité informatique peut se définir, d'un point de vue très général, comme ceci : « La sécurité informatique traite de la **prévention** et la **détection** des **actions non-autorisées** effectuées par les **utilisateurs** d'un système informatique ». Le but de la sécurité informatique est donc de s'arranger pour que les mauvaises choses ne se produisent pas…

Dans la définition précédente, nous parlons de « prévention » et de « détection ». Ces deux notions font parties du thème de la « sécurité » qu'elle soit informatique ou non. Concrètement, elles seront transposées dans notre système par une ou plusieurs solution(s) sécuritaire(s) particulière(s) à adopter en fonction de celui-ci. La notion de « réaction », non-évoquée dans la définition, est également importante car elle regroupe les mécanismes qui vont nous permettre de restaurer le système suite à une détection d'un problème.

Au niveau informatique, la définition évoque : les « actions non-autorisées » effectuées par les « utilisateurs ». Ces derniers correspondent aussi bien aux êtres

humains qu'aux processus informatiques (programmes) travaillant pour leur compte. Mais quelles sont ces actions potentiellement dangereuses que peuvent provoquer les utilisateurs d'un système informatique ? En fait, dans un tel système, les deux principaux agents sont « l'information » et « les entités de traitement de l'information ». L'information est généralement transmise par l'intermédiaire des « données ». Dans le cadre de ce mémoire, nous utiliserons les deux termes indépendamment, même s'il existe une nuance certaine. Les entités de traitement de l'information, d'un point de vue informatique, sont les ordinateurs comprenant l'ensemble de leurs ressources nécessaires à leur bon fonctionnement (CPU, mémoires, moyen de communication, etc…).

Afin de protéger nos deux agents, quatre grands principes à respecter sont souvent utilisés pour définir la sécurité informatique :

- **Confidentialité** : la prévention de la *divulgation non-autorisée* de l'information.
- **Intégrité** : la prévention de la *modification non-autorisée* de l'information.
- **Disponibilité** : la prévention du *refus non-autorisé* de l'accès à l'information ou aux ressources d'un système.
- **Identification** : la *détermination de l'identité* des acteurs de notre système informatique.

Ces quatre principes ne sont pas forcément indépendants suivant les techniques utilisées pour les préserver ou les compromettre. Par exemple, une attaque visant l'intégrité des données d'un système, pourra commencer par rendre un autre système indisponible afin d'obliger le transfert d'une information confidentielle par un mécanisme moins sécurisé et, enfin, exploiter cette information confidentielle pour accéder et modifier les données du système initialement visé.

Au cours des années, la sécurité informatique a évolué pour passer des simples postes isolés ou connectés par une liaison directe à un « mainframe », aux ordinateurs interconnectés les uns aux autres par l'intermédiaire de réseaux beaucoup plus élaborés. La sécurité à apporter s'en complique davantage de par l'augmentation du nombre d'accès potentiels à notre système par de multiples utilisateurs. De plus, suivant son fonctionnement initial, l'information échangée sur un réseau, comme Internet, ne présente aucune certitude quant au respect des quatre principes cités ci-dessus. En effet, le flux des données à transmettre est découpé en une suite de paquets de bits transmis sur le réseau et pouvant exploiter des chemins différents et inconnus à priori pour atteindre leur destination.

C'est pour cela que, dans ce travail, nous considérerons la sécurité informatique dans son sens le plus large. Nous ne ferons plus de distinction entre sécurité, sécurité des systèmes distribués et sécurité des réseaux.

D'autres grands principes, liés indirectement aux quatre autres et à l'adjonction des réseaux, peuvent être ajoutés à la liste :

- **L'audit** des événements produits.
- Le **contrôle d'accès** (autorisation) et **son application**.
- **L'administration** de la politique de sécurité.
- **La non-répudiation** des messages envoyés ou reçus.
- …

3.2 Outils sécuritaires : La cryptographie et le chiffrement

Le chiffrement est une technique très importante pour sécuriser une communication entre deux individus. Celle-ci consiste à transformer, à l'aide d'un algorithme, un élément de donnée quelconque appelé « texte clair » en un élément de donnée inintelligible appelé « texte chiffré » (vu comme une suite de bits aléatoires). L'opération doit bien entendu être réversible par la partie concernée. Les algorithmes de chiffrement utilisent un paramètre d'entrée appelé « clé ». Remarquons qu'un algorithme de déchiffrement peut utiliser une autre clé. Nous parlons donc ainsi de « clé de chiffrement » et de « clé de déchiffrement ». Cette clé correspond à un nombre entier. Il est important de souligner que la clé constitue un secret à garder par la ou les partie(s) visée(s). L'algorithme en lui-même n'est pas un secret ; il est même normalement largement publié.

Les techniques de chiffrement les plus courantes exploitent une combinaison de substitution et de transposition pour rendre l'information de départ vide de sens.

Parmi l'ensemble des méthodes de chiffrement, nous pouvons en distinguer deux catégories :

- Le chiffrement symétrique
- Le chiffrement asymétrique

3.2.1 Le chiffrement symétrique

Les algorithmes symétriques utilisent une clé pour chiffrer et une autre pour déchiffrer les messages échangés. Celles-ci s'obtiennent facilement l'une à partir de l'autre. Ces clés sont souvent dénommées « clés secrètes » (K_S). C'est pour cela que nous parlons également d'algorithmes à clés secrètes. Nous avions précédemment dit que la clé se présente sous la forme d'un nombre entier. Comme toute représentation de nombres entiers dans un système informatique entraîne une borne inférieure et supérieure, nous constatons que, dans le cas du chiffrement symétrique, il y a un ensemble fini de clés possibles pour une représentation de clé donnée.

Les algorithmes symétriques les plus connus sont : le Data Encryption Standard (**DES**), son extension le **Triple-DES**, le tout récent standard **AES** (**Rijndael**), etc…

La force d'un algorithme à clé secrète réside sur trois points :

(1) Sa capacité à préserver le texte chiffré suffisamment **vide de sens** pour quelqu'un ne possédant pas la clé de déchiffrement.

(2) Son **nombre suffisant de clés potentielles** à essayer. Ceci afin d'éviter une attaque de force brute (par essai de toutes les combinaisons possibles).

(3) La **préservation du secret** (la clé secrète) par les deux parties de la communication !

La technique de chiffrement à clé secrète nous permet de préserver la confidentialité des messages. Si nous la combinons à d'autres notions comme les codes MAC (voir plus loin), nous pouvons obtenir la garantie absolue de vérification de l'intégrité des messages.

D'un point de vue pratique, le chiffrement symétrique est utilisé pour le chiffrement massif du flux d'information échangé.

Remarque : Si deux personnes cherchent à s'échanger des messages chiffrés par la technique des clés symétriques mais qu'elles ne possèdent pas cette clé secrète en commun, une solution consiste à passer par une entité tierce de confiance qui fournira, par un moyen sécurisé, la clé secrète recherchée. Il existe cependant d'autres techniques permettant cet échange de clés sans intermédiaire de confiance.

3.2.2 Le chiffrement asymétrique

Les algorithmes à chiffrement asymétrique appelés généralement à clé publique, utilisent deux clés qui ne se déduisent pas facilement l'une de l'autre. Nous avons ainsi la clé privée (K_C - pour clé confidentielle) et la clé publique (K_P). Chacune d'elle peut servir à chiffrer ou à déchiffrer. Mais les objectifs sécuritaires résultant ne sont pas les mêmes. Chaque partie désirant communiquer doit posséder un couple « clé publique - clé privée ». Par définition le secret à garder correspond à la clé privée tandis que la clé publique peut être librement divulguée aux correspondants. Les données chiffrées par une clé publique ne peuvent être déchiffrées que par sa clé privée correspondante et vice-versa. Cette technique simplifie la distribution des clés car l'autre partie ne doit pas connaître la clé privée correspondante, ce qui est l'inverse du chiffrement symétrique.

La force d'un algorithme cryptographique à clé publique réside sur quatre points :

(1) Sa capacité à préserver le texte chiffré suffisamment **vide de sens** pour quelqu'un ne possédant pas la clé de déchiffrement .

(2) Son **nombre suffisant de clés potentielles** à essayer. Ceci afin d'éviter une attaque par force brute (par essai de toutes les combinaisons possibles).

(3) La **préservation du secret** (la clé privée K_C) par la partie concernée !

(4) L'**impossibilité de déduire** une clé privée (K_C) à partir de sa clé publique (K_P).

Parmi les algorithmes actuels exploitant la technique des clés publiques, nous avons **RSA** inventé par Rivest, Shamir et Adleman.

Afin de comprendre les différents cas d'utilisation de la technique des clés publiques, introduisons quelques notations ainsi que deux personnes bien connues dans le monde de la cryptographie : Alice et Bob. Dans la littérature cryptographique, Alice et Bob tentent de s'envoyer des mots doux sans que personne d'autre ne puisse les lire et les comprendre. Pour notre exemple, imaginons que Bob possède une clé publique « K_P » et une clé privée « K_C ». Pour simplifier nous supposons qu'Alice n'en possède pas. Alice représente ici n'importe quelle personne désirant communiquer avec Bob. La fonction E (K, m) permet de chiffrer un message « m » à l'aide d'une clé « K » qu'elle soit privée ou publique et d'obtenir un texte chiffré « c ». Nous avons son

symétrique D (K, c) qui déchiffre le texte chiffré « c » et nous fournit le message clair initial « m ».

Nous constatons deux cas :

- Alice envoie un message chiffré à l'aide de la clé publique de Bob :

  Alice : $E(K_P, m) = c$ → Bob : $m = D(K_C, c)$

  Toute personne possédant K_P, peut envoyer un message chiffré à Bob et seulement lui, sera en mesure de le déchiffrer. Nous préservons de cette manière la confidentialité du message envoyé.

- Bob envoie un message chiffré à l'aide de sa clé privée à Alice :

  Bob : $E(K_C, m) = c$ → Alice : $m = D(K_P, c)$

  Toute personne possédant K_P est en mesure de déchiffrer le message envoyé par Bob. Nous assurons l'identification de l'origine du message car Bob est le seul à pouvoir émettre les messages chiffrés à l'aide de sa clé privée (voir aussi notion de signature).

Par l'adjonction de résumés de message (voir plus loin), nous pouvons également garantir l'intégrité totale des messages échangés.

3.2.3 Techniques hybrides

La méthode hybride consiste à combiner les deux catégories de chiffrement précédentes. La raison provient d'un facteur de performance car le chiffrement et le déchiffrement à clé secrète sont de cent à mille fois plus rapides que ceux à clé publique pour une sécurité comparable. L'idée est donc d'exploiter un algorithme à clé publique pour s'échanger une clé secrète commune. Celle-ci est ensuite utilisée pour chiffrer le flux massif d'information transmis.

3.2.4 Résumé de message (Message Digest)

Résumé de message sans clé (MIC - Message Integrity Codes)

Un résumé de message, hachage cryptographique, empreinte numérique ou somme de contrôle cryptographique est une suite de bits de taille fixe représentant un ensemble

de données qui peut être d'une taille arbitraire. Les résumés de message sont les analogues cryptographiques des codes correcteurs d'erreurs utilisés dans le monde de la transmission de données (code CRC par exemple). Ils ont cependant des garanties différentes des codes correcteurs d'erreurs : le caractère unidirectionnel et la résistance aux collisions. Le caractère unidirectionnel nous assure qu'il n'est pas possible de retrouver le message original à partir d'un résumé de message. La résistance aux collisions nous assure qu'une petite modification dans le message original, entraîne de grandes modifications dans le résumé de message. Autrement dit, il est très difficile de trouver un autre message tel que nous obtenons le même résumé de message.

D'un point de vue pratique, les deux standards les plus utilisés sont : Message Digest (MD5, MD4, MD2) et Secure Hash Algorithm-1 (SHA-1). Le standard SHA-1 qui crée un résumé de message sur 160 bits est une amélioration du MD5 qui le fait sur 128 bits.

Associés à la cryptographie à clé publique, les résumés de message offrent un moyen de signer des documents numériques et de garantir l'intégrité des messages échangés ainsi que la non-répudiation de l'émetteur.

Résumé de message à clé secrète (MAC – Message Authentication Code)

Les code MAC sont des résumés de message mais constitués à l'aide d'une clé secrète (chiffrement symétrique). Ils fondent leur sécurité sur l'algorithme à clé secrète utilisé et sur le maintien du secret (la clé) par nos deux parties. La clé secrète nécessaire au chiffrement du flux de données est également souvent utilisée pour créer le code MAC correspondant au bloc de données transmis. Mais il vaut mieux éviter de mélanger les clés pour un usage différent. Cette technique utilisée pour les messages chiffrés par clé secrète nous garantit l'intégrité des messages transmis.

Remarque : Le chiffrement à clé secrète seule garantit uniquement la confidentialité du message. Une personne intermédiaire a toujours la possibilité de modifier le flux transmis même si au bout du compte le message déchiffré paraît totalement incohérent. En particulier, si le message clair est aléatoire, comme la transmission d'une clé par exemple, le destinataire n'a aucun moyen de vérifier l'intégrité de l'information transmise sans un résumé de message.

Les méthodes créant des résumés de message, qu'ils soient à clé secrète ou sans, sont souvent appelées « fonction de hashage ». Nous utilisons la notation « h » pour un résumé de message, « H » pour la fonction de hashage et « m » pour le message à résumer : $h = H(m)$.

3.2.5 Signature numérique

En combinant les résumés de messages et la technique des clés asymétriques, nous avons la possibilité de créer une signature numérique avec appendice. Il existe également d'autres techniques pour créer des signatures numériques mais intéressons-nous plus particulièrement à la première. Il n'est pas toujours nécessaire de chiffrer le contenu total d'un message dans le cas, par exemple, où le respect de la confidentialité n'est pas capital. Par contre, il peut être primordial d'assurer l'intégrité du message (n'a pas été modifié), l'identification de l'origine du message (déterminer son émetteur) et la non-répudiation (son émetteur ne peut pas nier l'avoir envoyé). Nous obtenons ce résultat en créant un résumé de message de l'information à envoyer puis en le chiffrant à l'aide de notre clé privée. Le destinataire peut alors déchiffrer le résumé de message grâce à la clé publique de l'émetteur et comparer le résumé de message reçu avec celui qu'il vient de calculer à partir du message reçu. Il faut bien entendu utiliser la même fonction de hashage.

Les algorithmes de signature pouvant être différents des algorithmes de chiffrement à clé publique, nous parlons donc de **signer** un message et de **vérifier** sa signature. Introduisons également ces notations : « s » est la signature d'un message « m » et « S » est la méthode de signature : $s = S(K_C, m) = E(K_C, H(m))$

DSA (Digitial Signature Algorithm) et RSA sont des exemples pratiques d'algorithmes capables de créer des signatures numériques.

3.2.6 Certificats numériques

Nous sommes de plus en plus amenés à devoir communiquer à travers un réseau de télécommunication. Celui-ci a pour désavantages d'occulter les signes permettant de reconnaître la personne prétendant être notre correspondant et de n'offrir aucune garantie quant à la confidentialité et l'intégrité des messages échangés. De plus, nous reconnaissons le plus souvent nos partenaires par des informations significatives pour l'être humain comme, par exemple, un nom. La cryptographie à clé publique peut nous aider dans la tâche d'authentification de nos interlocuteurs (voir protocoles d'authentification). Mais pour cela, chaque personne doit posséder une clé publique, elle-même distribuée à chacune des personnes désirant le contacter. Cette hypothèse n'est pas toujours accomplie et nous ne possédons pas toujours un canal sécurisé pour transmettre cette clé. Il faut donc trouver un moyen pour associer de manière fiable une identité à une clé publique.

C'est dans ce contexte que nous parlons de certificats numériques. Celui-ci correspond à un document précisant cette association auquel nous avons adjoint la signature numérique de son contenu effectuée par une tierce personne de confiance

(**TTP** : Trusted Third Party). Le certificat n'a de la valeur pour une personne que si elle a confiance en cette tierce personne. Par son acte de signature, elle affirme que l'identité X a pour clé publique K_P. Evidemment, la clé publique de la personne de confiance est préalablement connue de façon à pouvoir vérifier la signature. Ce principe nous autorise à ne connaître, à l'extrême, que la clé publique de notre personne de confiance. Dans cet exemple, nous avons parlé d'individus qui communiquent sur un réseau. Mais le procédé peut s'étendre à n'importe quel type d'entité que ce soit des personnes, des programmes, des composants électroniques, des droits d'accès, … Ce qui compte, c'est d'associer une clé publique à une information quelconque.

Nous l'avons indirectement souligné, le certificat se base sur la notion de confiance. Ainsi, si nous lui attribuons une certaine transitivité, nous pouvons étendre le principe et former une chaîne de certificats. « Les amis de mes amis sont mes amis… ». Dans le jargon de la sécurité, chaque entité de confiance fournissant des certificats est appelé **CA** (Certificate Authority). Celles-ci peuvent être hiérarchisées et former une structure en forme d'arbre. Chaque CA étant représenté par un nœud de l'arbre, a confiance en son nœud directement supérieur (son père) et possède sa clé publique. Les CA situés au niveau des feuilles de l'arbre sont les entités directement en contactes avec les personnes désirant communiquer entre elles. Pour que deux individus n'ayant pas le même CA comme tiers de confiance, puissent acquérir leur clé publique respective, il faut trouver un nœud de l'arbre en commun et leur fournir la chaîne de certificats correspondant au chemin reliant leurs CA dans l'arbre. Evidemment, le CA racine de l'arbre, ne possède pas de certificat affirmant la confiance qu'il a en son père puisqu'il n'en a pas. Il « auto-signera » donc son propre certificat.

Il existe actuellement deux formats principaux de certificats numériques qui n'établissent pas la confiance de la même manière.

Le format X.509

Celui-ci se base sur le principe de la « Certificate Authority » décrit dans le paragraphe précédent.

Le format PGP

Le format PGP (Pretty Good Privacy) procède de façon différente : la notion de confiance se base sur le nombre de personnes de confiance affirmant la correspondance « clé publique – identité d'une personne ».

Dans la terminologie des certificats numériques, nous parlons : d'émetteur, de sujet, de période de validité, de révocation de certificats, … Normalement, un certificat est créé par un « émetteur ». Ce-dernier signe le certificat qui concerne une identité appelé « sujet ». Un certificat possède également une période de validité qui peut aller d'un mois à plusieurs années. Après cette période, le certificat expire. Un certificat révoqué est, quant à lui, un certificat qui a perdu sa validité avant sa date d'expiration. Les causes sont par exemple : une clé compromise, un algorithme de signature devenu cassable, le porteur du certificat change de clé publique, … Il est donc important de pouvoir informer les parties intéressées de la révocation de ces certificats. Ceci s'accompli généralement par une liste de révocation de certificats (CRL : Certificate Revocation List) disponibles, par exemple, dans un annuaire informatique.

3.2.7 Notion de protocoles

La cryptographie seule permet d'assurer la confidentialité, l'intégrité et la non-répudiation de l'émission d'un message. Mais il n'en est rien pour l'identification (éventuellement mutuelle) de personnes en communication et pour la non-répudiation de la réception de messages. De plus, dans le cas de la cryptographie symétrique, il n'existe pas toujours une entité tiers de confiance pour fournir à deux personnes, la clé secrète qu'elles doivent utiliser pour communiquer.

Les protocoles sont là pour résoudre ce genre de problème. Un protocole est une suite ordonnée de messages échangés entre deux entités dans le but de conclure un certain résultat. Parmi ceux-ci, nous citons entre autres : les protocoles d'authentification, de non-répudiation et d'échange de clé.

3.2.8 Sécurité au niveau des couches « réseau » et « transport »

L'infrastructure PKI Secure Socket Layer (**SSL v3**) et le récent standard Transport Layer Security (**TLS**), très semblables, servent à envoyer des informations de manière sécurisée sur Internet, notamment pour le courrier électronique, le FTP (File Transfer Protocol) et les réseaux VPN (Virtual Private Network). Il s'agit plus précisément de protocoles complémentaires au protocole TCP de la couche « transport » du modèle réseau « TCP/IP » et qui implémentent trois grandes garanties cryptographiques :

- l'authentification (d'une des deux parties)
- la confidentialité et l'intégrité des messages
- l'échange sécurisé de clés secrètes

La non-répudiation ne fait pas partie de ces garanties. De plus, dans sa version standard, seul le serveur doit s'authentifier auprès du client. Mais une authentification mutuelle est évidemment réalisable.

Le protocole SSL/TLS suit différentes étapes pour effectuer une transaction sécurisée : la négociation des paramètres, l'échange de certificat(s) numérique(s), l'accord sur les clés secrètes, l'authentification et le chiffrement des données.

Le protocole **IPsec** cherche à atteindre le même genre de garanties cryptographiques que SLL et TLS. Sa grande différence provient du fait qu'il est destiné à remplacer la couche « réseau » (la couche IP) du modèle « TCP/IP ». Ceci nécessite évidemment une compatibilité certaine entre les différents routeurs constituant le réseau. Il existe d'autres différences, mais nous n'allons pas plus loin dans le cadre de ce travail.

3.3 Outils sécuritaires : Les contrôles d'accès

La cryptographie nous offre un mécanisme fiable pour gérer la confidentialité et l'intégrité des données. Mais n'oublions pas qu'elle repose tout du moins sur une hypothèse principale : la préservation du secret (la clé). Il est souvent plus facile de chercher à obtenir les clés directement là où elles sont entreposées que de s'attaquer à un système cryptographique bien conçu. La cryptographie ne résout pas non-plus notre problème sécuritaire du contrôle de l'accès aux ressources d'un système informatique.

Il s'agit donc de mettre au point un mécanisme capable de contrôler ce que fait quelqu'un sur les ressources du système. Pour être plus général, nous parlons de « sujets » agissant par l'intermédiaire « d'actions » sur des « objets ». Par exemple : un utilisateur X (sujet) lit (action) des données dans un fichier (objet). Le contrôle s'effectue le plus souvent sur base d'une matrice de contrôle d'accès le plus souvent implémentée sous la forme d'une liste des permissions accordées à un sujet (ACL : Access Control List). Par défaut, toute permission d'accès lui est refusée sauf celles indiquées dans la liste.

Une bonne question consiste à se demander où pouvons-nous placer ce contrôle dans un système informatique ? La réponse dépend essentiellement du type d'architecture utilisée. Dans la plupart des cas, la première couche logicielle capable de distinguer un sujet d'un objet est le système d'exploitation. La majorité des OS actuels sont conçus pour fournir un contrôle d'accès aux ressources sur base de l'utilisateur et des droits qu'un administrateur lui a octroyés (par exemple : UNIX, Windows NT, …). La

plate-forme Java 2 que nous étudions dans le prochain chapitre est, elle aussi, en mesure de contrôler les permissions (d'action) attribuées aux sujets.

Mais à l'heure des réseaux et de l'interopérabilité de systèmes hétérogènes, peut-être devons-nous aussi penser à implémenter ce genre de mécanisme un peu plus haut dans les couches logicielles ? En fait, cette question vise les problèmes d'authentification des sujets et d'administration de leurs droits dans un système distribué. Nous verrons plus loin que, dans le cadre de Jini, des solutions particulières ont étés implémentées pour mettre au point « une architecture sécuritaire pour Jini ».

Il est donc très important de réaliser que pour atteindre nos objectifs de sécurité, il est nécessaire d'utiliser une cryptographie résistante à ses attaques en conjonction de mécanismes de contrôle d'accès aux ressources fiables.

Chapitre 4

Architecture sécuritaire de Java

Dans ce chapitre, nous détaillons les mécanismes et les outils sécuritaires de la plate-forme Java 2 qui sont probablement exploitables dans l'actuelle implémentation de Jini. Enfin, à titre de comparaison envers Jini, nous parlons du « service sécurité » de l'architecture CORBA. Pour plus de détails sur l'architecture sécuritaire de la plate-forme Java 2, nous nous référons à [11].

4.1 Sécurité de la plate-forme Java 2

La plupart des travaux concernant la sécurité de la plate-forme Java 2 se sont focalisés sur la protection de la « **Java Virtual Machine** » (**JVM**) des attaques potentielles d'un code malicieux (en général téléchargé depuis le réseau). Cette attention toute particulière vient du fait que la plate-forme autorise l'exécution des « **applets** ». Ceux-ci sont de petits programmes Java généralement téléchargés dans la JVM d'un navigateur Internet pour accomplir des tâches comme : l'animation de composants graphiques, la maintenance d'informations, le calcul de certains résultats avant retransmission de ceux-ci vers le serveur dont l'applet est originaire.

Avant de s'attaquer aux mécanismes mis au point par les développeurs de chez Sun Microsystems, posons-nous deux questions : Que représente réellement la plate-forme Java 2 ? Comment fonctionne-t-elle ?

4.1.1 Fonctionnement de Java

Traditionnellement, Java est défini comme un langage interprété orienté objet à typage fort. Mais Java est avant tout une plate-forme logicielle comprenant trois concepts importants :

- Le langage Java.
- La Java Virtual Machine (JVM).
- Un API de librairies.

Avant d'exécuter un programme Java, composé d'une ou de plusieurs classe(s) d'objet(s), il doit être préalablement compilé dans un langage machine intermédiaire du nom de « **bytecode** ». Celui-ci est destiné à être exécuté dans la JVM qui pourrait finalement être comparée à un processeur dont le jeu d'instruction est celui du bytecode. La JVM est dépendante de la plate-forme sous-jacente sur laquelle elle est exécutée. Il s'agit le plus souvent d'un système d'exploitation classique. Mais ceci ne l'empêche pas d'être implémentée directement pour tourner dans un appareil embarqué ou même être totalement conçu en hardware et donc devenir par la même occasion un microprocesseur à part entière. Remarquons que grâce à la JVM, le bytecode (et donc Java) devient un langage machine indépendant de toute plate-forme matérielle et/ou logicielle sous-jacente. Cette indépendance de plate-forme est la condition nécessaire au principe du « code mobile ».

Comme nous l'avons dit un peu plus haut, la plate-forme Java 2 comporte trois concepts différents. Il n'est dès lors pas étonnant que la sécurité apparaisse à chacun de ces niveaux.

4.1.2 Sécurité au niveau du langage

Beaucoup d'attaques se basent sur l'acquisition d'informations confidentielles situées dans une zone précise de la mémoire. Des langages de programmation comme C/ C++ autorisent potentiellement le développeur à accéder à n'importe quelle zone mémoire par l'intermédiaire de pointeurs. Dépasser les limites d'un vecteur pour arriver sur un espace mémoire qui n'appartient pas au programme exécuté est une erreur classique, souvent involontaire mais commise par les programmeurs de ce langage. D'autres attaques, comme l'accès ou la modification d'espaces mémoire normalement réservés, entraînent l'instabilité immédiate du système.

Le langage Java a été conçu pour protéger le « type » des données qu'il manipule. Il impose d'ailleurs les restrictions suivantes : pas d'arithmétique de pointeur ; pour toute variable ou pour tout objet, il ne peut y avoir de valeur indéfinie ; la libération

de l'espace mémoire alloué aux objets Java est gérée automatiquement par le « **Garbage Collector** » ; il n'est pas autorisé de dépasser les limites d'un vecteur ; il n'autorise pas le « casting » illégal des objets ; il contrôle la visibilité et l'accès des parties membres d'une classe Java.

Ceci signifie, par exemple, qu'aucun programme Java ne peut référencer un objet en utilisant un type incorrect, ni référencer une zone de mémoire non-allouée précédemment, ni construire des pointeurs « à la main ». De même, le langage étant orienté objet, il offre les restrictions d'accès habituels sur les classes, les méthodes et les attributs comme : « public, private, protected ». Il étend même la notion de restriction au niveau du « package » correspondant au répertoire regroupant un ensemble de classes.

La majorité de ces vérifications est réalisée par le compilateur Java qui transforme le code source en bytecode. Mais le bytecode peut très bien être écrit manuellement, sans passer par le compilateur. Il faut dès lors le vérifier avant de charger l'objet correspondant dans la machine virtuelle pour exécution.

4.1.3 Sécurité au niveau de la JVM

Le vérificateur de bytecode

Chaque classe Java est instanciée sous la forme d'objets par l'intermédiaire d'un organe particulier appelé « **ClassLoader** ». Celui-ci s'occupe de chercher, vérifier puis charger la classe dans la JVM. Ici, les vérifications sont statiques. En effet, elles sont réalisées avant que l'objet soit réellement instancié. Elles tentent de s'assurer que le bytecode de l'objet à charger respecte bien les hypothèses de base du langage Java. D'autres vérifications, dynamiques cette fois, ne peuvent être effectuées qu'au moment de l'exécution du programme. Nous citons comme exemples : la vérification de type, la vérification des limites d'un vecteur et l'interdiction d'accéder à une classe de base en tant que sous-classe de celle-ci.

Le « Security Manager » et le modèle sécuritaire du « Sandbox »

Comme nous l'avons déjà introduit plus haut, les applets et le code mobile qui découle de leur utilisation induisent des failles sécuritaires potentielles qu'il ne faut pas négliger. Par exemple, l'applet téléchargée ne doit pas pouvoir accéder à des informations confidentielles de l'utilisateur. Pour résoudre ce problème, la première version de Java (JDK 1.0) est apparue avec le modèle du « Sandbox » (cf. Figure 4.1). Celui-ci propose « d'enfermer » une applet dans un espace mémoire propre et de lui interdire tout accès aux ressources critiques du système. Il ne peut ainsi ni accéder aux informations confidentielles de l'utilisateur, ni gêner l'exécution des autres processus

du système. Cependant, l'applet a la permission de contacter son serveur de provenance. Le mécanisme distingue clairement le code venu du réseau et considéré comme non-fiable et à enfermer, du code local qui lui, se voit librement accéder à toutes les ressources du système. L'organe contrôlant la bonne application de cette distinction s'appelle le « **Security Manager** ».

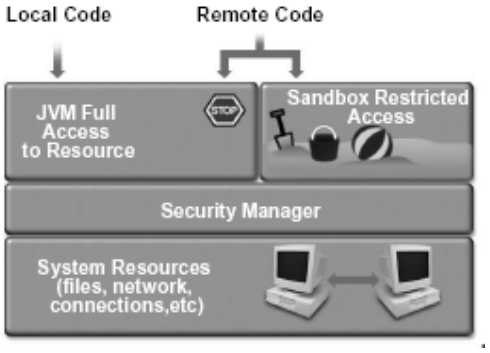

Figure 4.1 : Modèle sécuritaire du « Sandbox » (JDK 1.0)

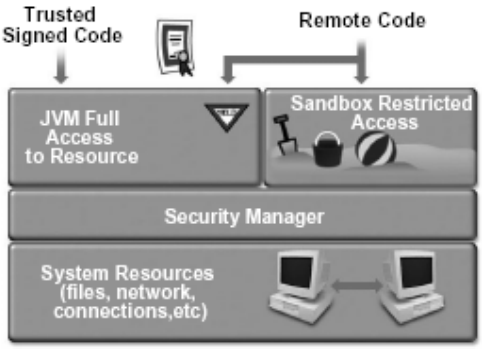

Figure 4.2 : Modèle sécuritaire du « Sandbox »(JDK 1.1)

Le JDK 1.1 apporte la notion de signature numérique (cf. Figure 4.2). Chaque programme Java est caractérisé par un ensemble d'objets dont l'implémentation est stockée dans des fichiers « **class** ». Ceux-ci peuvent être totalement intégrés dans un fichier compressé du nom de fichier « **jar** ». L'auteur d'une applet dispose alors de la possibilité de signer numériquement le fichier « jar » contenant son application, puis d'insérer cette signature dans le fichier compressé en question. Grâce à ce mécanisme, la plate-forme est à même de distinguer le code téléchargé certifié (par la signature numérique qui est vérifiée) du code téléchargé non certifié qui lui doit rester enfermer dans son sandbox. Mais la notion de code local peut entraîner quelques ambiguïtés. Par exemple un code présent dans un répertoire NFS (Network File System) « monté » dans notre système de fichier, doit-il être considéré comme local ou bien distant ?

Depuis le JDK 1.2, et donc la création de la plate-forme Java 2, le Security Manager de toute JVM a la possibilité d'effectuer un contrôle fin sur toutes les opérations réalisées par un programme Java (cf. Figure 4.3). Ce mécanisme s'applique aussi bien au code local qu'au code venu du réseau.

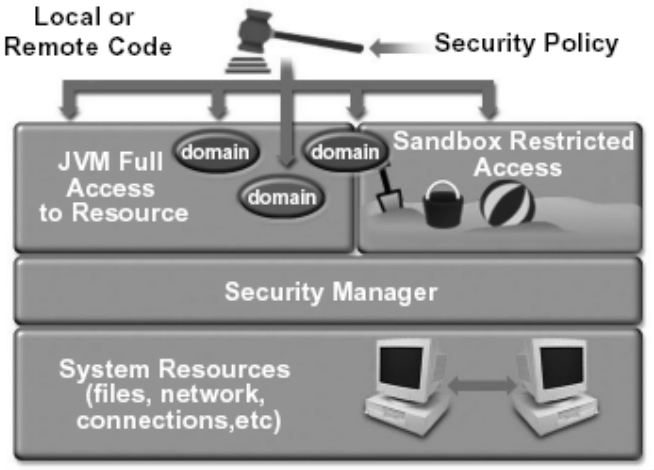

Figure 4.3: Modèle sécuritaire de la plate-forme Java 2

Notions de permissions et de domaines de protection

Pour représenter les droits d'accès vis-à-vis des ressources du système, Java utilise la classe « **`Permission`** ». Celle-ci associe une « cible », comme une classe représentant une ressource, à un ensemble d'actions qui peuvent lui être appliquées. Java introduit également la notion d'implication de permissions. Ensemble, ces deux notions forment la base de toutes les décisions de contrôle d'accès.

Prenons, par exemple, la classe « FilePermission ». Celle-ci a pour cible un fichier ou un répertoire et pour action : une opération de lecture (r), d'écriture (w), d'effacement (d), d'exécution (x). Par la notion d'implication, nous affirmons les deux lignes suivantes :

- FilePermission(/tmp/*, r) → FilePermission(/tmp/abc.txt, r)
- FilePermission(/tmp/-, rw) → FilePermission(/tmp/sub/abc.txt, r)

Dans l'architecture sécuritaire ainsi présentée, nous assignons un ensemble de permissions à l'adresse réseau (**codebase**) en provenance du code et au nom de la personne censée avoir signé le code. Cette association correspond aux entrées d'un petit fichier texte de configuration sécuritaire du nom de « **policy file** ». Celui-ci peut être communiqué à la JVM lors de son exécution. La grammaire simplifiée d'une de ses entrées a la forme suivante :

```
grant [codebase url] [signedby signers] {
    {permission}*
}
```

Le fichier « policy » contient également la localisation du « **keystore** ». Celui-ci est le fichier ou la base de données contenant les clés privées, publiques ou secrètes de l'utilisateur.

Java introduit également la notion de « **domaine de protection** ». Celle-ci permet de regrouper un ensemble de classes ayant les mêmes permissions et le même codesource. Un **codesource** correspond à un codebase auquel nous avons ajouté un ensemble de clés publiques.

Remarque : Java permet également de joindre à ses fichiers « jar » des certificats numériques au format X.509.

4.1.4 Sécurité au niveau de l'API

Pour satisfaire les besoins cryptographiques ou d'authentification des applications Java, Sun Microsystems a ajouté, depuis le JDK 1.3, une architecture cryptographique se présentant sous la forme de « packages » supplémentaires au JDK. Ceux-ci font intégralement partie du tout récent JDK 1.4.

- JCE : Java Cryptography Extension. Il s'agit d'une librairie Java fournissant des primitives cryptographiques.
- JSSE : Java Secure Socket Extension. Il s'agit d'un package optionnel fournissant des sockets exploitant SSL ou TLS.
- JAAS : Java Authentication and Authorization Service. Il s'agit d'une extension ajoutant des mécanismes pour l'authentification des utilisateurs avec les contrôles d'accès appropriés.

4.1.5 Outils sécuritaires pratiques

- Keytool : il fournit un accès au keystore par l'intermédiaire de la ligne de commande.
- Jarsigner : il est utilisé, via la ligne de commande, pour signer des fichiers « class » dans un fichier « jar ».
- Policy Tool : il fournit une interface graphique pour créer des fichiers « policy ».

4.1.6 Quelques remarques

Le bytecode est interprété. Ceci présente un certain avantage car la JVM (le logiciel chargé de son interprétation) est en mesure d'effectuer des vérifications plus élaborées que celles qu'un processeur normal peut réaliser sur son code machine. Mais d'un autre coté, la JVM étant pour la plupart des cas un logiciel tournant sur un OS, elle est fortement dépendante de la sécurité offerte par celui-ci. Les fichiers critiques de la JVM sont-ils correctement protégés d'une modification éventuelle ? En est-il de même pour les informations confidentielles comme les clés secrètes ? Il est important le souligner car Jini est une couche logicielle entièrement dépendante de Java : il hérite fortement de toutes les forces et faiblesses des couches logicielles le précédant.

4.2 Sécurité de l'architecture CORBA

Dans cette section, nous énumérons brièvement les fonctionnalités de l'infrastructure sécuritaire de CORBA [18]. Nous explicitons également certains de ses mécanismes de base. L'infrastructure sécuritaire de CORBA comprend la définition d'un service « sécurité », lequel inclut essentiellement les fonctionnalités suivantes :

- **L'identification et l'authentification** des « principals » (utilisateurs et serveurs) ; la génération des « credentials » (crédits) des utilisateurs (qui sont des certificats de leurs droits) ; la délégation de ces crédits.

- **L'autorisation et le contrôle d'accès** : ils peuvent être effectué sur les objets CORBA quand ils reçoivent une invocation de méthode (à l'aide d'une ACL, par exemple).

- **L'audit de sécurité** : de l'invocation des méthodes, sur les serveurs.

- **Les communications sécurisées** : informations chiffrées, …

- **La non-répudiation** : Le serveur crée et stocke des crédits qui prouvent que l'invocation de la méthode a été faite pour le compte de tel ou tel utilisateur.

- **L'administration** : pour gérer l'ensemble du service.

Le service sécurité permet aux utilisateurs d'acquérir leurs crédits individuels sur base d'une authentification préalable (comme un mot de passe, par exemple). Pour garantir une sécurité correctement appliquée aux invocations de méthodes distantes, le service « sécurité » nécessite une étroite coopération avec l'ORB. Pour effectuer une communication sécurisée, les crédits de l'utilisateur sont envoyés avec chaque requête

effectuée vers un objet CORBA distant. Quand celle-ci arrive chez le serveur, celui-ci vérifie qu'il s'agit d'un client possédant les droits requis pour invoquer cette méthode (en vérifiant si le certificat est signé par une entité de confiance, par exemple). La décision est prise en consultant un objet qui possède une ACL.

Nous constatons que l'infrastructure sécuritaire offerte par CORBA recouvre un domaine plus large que la simple sécurité des réseaux (comme, par exemple, celle de l'infrastructure sécuritaire du modèle OSI [19], [20]) ne fut-ce que par l'utilisation des termes « autorisation » et « administration ».

Il s'agit bien ici d'un modèle sécuritaire destiné à une infrastructure pour objets distribués. Ce modèle semble donc adéquat pour d'autres infrastructures comme Jini ou RMI. Cependant, nous avons déjà souligné dans un chapitre précédent (section 2.4.5) les nombreuses différences séparant CORBA de ces deux dernières. Tout au long des prochains chapitres nous découvrirons les nouveaux problèmes sécuritaires issus de ces distinctions.

Chapitre 5

Analyse des besoins en sécurité de Jini

Nous avons précédemment détaillé le fonctionnement de Jini ainsi que les mécanismes de sécurité offerts par la plate-forme Java 2. Dans ce chapitre, nous allons tout d'abord souligner les mécanismes sécuritaires disponibles dans l'actuelle version de Jini. En particulier, nous montrons qu'ils sont inexistants et que certains mécanismes visant à résoudre les problèmes sécuritaires liés au code mobile, comme le modèle du « Sandbox », ne sont pas adéquats aux applications Jini. Ensuite, nous procédons à une analyse rigoureuse des différentes interactions entre les principaux acteurs d'un réseau Jini. Cette analyse porte sur des considérations de bas et de haut niveau. Pour terminer, nous élaborons une liste d'exigences sécuritaires applicable à la technologie Jini et qui nous servira de point de référence pour comparer différentes architectures sécuritaires présentées dans un prochain chapitre. A chacune de ces exigences, nous proposons également quelques éléments de réponse ainsi que les difficultés liées à la mise en place de ces solutions.

Remarque : Les liste des différents scénarios de notre analyse de haut niveau ainsi que la liste des exigences sécuritaires de la fin de ce chapitre tentent toutes de recouvrir au mieux les différents cas d'attaques imaginables à leur niveau. Néanmoins, par soucis de précision, nous ne pouvons considérer qu'elles sont exhaustives.

5.1 Sécurité actuelle de Jini

Jini a été initialement conçu de façon à bien faire ce pour quoi il a été créé : la découverte de services, l'administration automatique des réseaux dynamiques spontanés, la tolérance aux pannes, … les mots d'ordre furent donc « **fiabilité** et

simplicité ». Il ne faut pas oublier que Jini fut à son origine une technologie d'essai et ce n'est que par après qu'elle trouva une vocation directe dans les réseaux d'appareils électroniques. D'un point de vue commercial, le but initial de Sun Microsystems fut d'être le premier sur le marché. Il n'est pas étonnant dès lors que la sécurité ait été reportée à une seconde mouture du software.

Au niveau technique, Jini est écrit en Java et est donc tout destiné à être exécuté dans une JVM. Il hérite de ce fait des mécanismes sécuritaires offerts par celle-ci.

Parmi l'ensemble des questions que nous sommes en droit de nous poser concernant la sécurité de Jini, il y en a une toute particulière qui sort du lot : « Pourquoi devons-nous parler de sécurité pour l'architecture Jini alors que Sun Microsystems nous affirme que Java est une plate-forme sécurisée et robuste pour développer des applications Internet ? »

Après étude des différents mécanismes de sécurité de la plate-forme Java 2, nous constatons qu'elle est suffisamment sécurisée et robuste pour développer des applications comme les « applets », car il s'agit bel et bien de ce type d'application « Internet » auquel Sun fait référence. Evidemment, cette affirmation n'a de sens que dans la mesure où les mécanismes sécuritaires sont bien configurés par l'utilisateur. Ainsi, par exemple, doit-il protéger l'accès à ses clés secrètes. Celles-ci se trouvent généralement dans un fichier résidant sur le disque dur de l'utilisateur. Les droits d'accès sont configurés par un administrateur ou l'utilisateur lui-même et sont contrôlés par l'OS qui doit disposer d'un système de fichier permettant ces vérifications. De même, l'utilisateur doit y avoir préalablement placé les clés publiques des organismes de validation de certificat (CA) afin de pouvoir contrôler la signature éventuellement présente dans une applet téléchargée. Le « SecurityManager » de la JVM, doit normalement être aussi en contact avec un serveur de révocation de certificats. Enfin, l'utilisateur doit dans certain cas configurer le fichier de politique de sécurité pour donner certains droits d'accès supplémentaires aux ressources de son système pour que certaines applet puissent, par exemple, effectuer un travail de maintenance.

Si nous analysons l'utilisation habituelle des **applets**, nous constatons qu'elles peuvent être considérées comme « **de petites applications Web limitées** ». Elles ont en effet habituellement pour rôle de gérer les animations graphiques au sein des pages Web ou bien d'effectuer quelques calculs préliminaires sur des formulaires avant transmission du résultat vers le serveur d'origine. Le modèle sécuritaire du « Sandbox », visant à enfermer l'applet dans un espace mémoire propre, isolé de tous les autres composants du système et n'ayant pour seule porte de sortie que la

communication réseau vers son serveur de provenance, semble être un modèle sécuritaire approprié.

Lorsque nous parlons de Jini et du code mobile qui découle de l'utilisation d'un proxy, nous parlons de véritables applications distribuées nécessitant un champ d'action plus large que celui nécessaire aux applets. Dans le monde Jini, l'application serveur (le service) est constituée de l'agrégation du proxy (exécuté chez le client) et du service (exécuté sur le serveur). De même, l'application cliente est constituée du programme créé par le client et d'un ensemble de proxy utilisés pour dialoguer avec le(s) serveur(s). Un seul proxy peut en effet correspondre avec plusieurs serveurs différents et nécessiter par la même occasion plusieurs connexions réseau différentes. Nous avons donc dans le cas de Jini un ensemble d'applications distribuées nécessitant des droits d'accès aux ressources du client beaucoup plus importants !

Si nous nous penchons sur des infrastructures pour objets distribués tels que CORBA ou RMI, nous constatons que les mécanismes sécuritaires sont essentiellement placés du coté du serveur. En effet, c'est lui qui effectue le travail pour le compte du client et c'est donc lui qui a intérêt à contrôler fortement ce qu'on lui demande de faire. Le client n'a normalement pas à se soucier d'un potentiel corps étranger venant s'exécuter sur sa machine. Remarquons que, pour RMI, le code mobile existe, ne fut ce que pour le téléchargement dynamique du « stub » correspondant à l'objet remote utilisé. Pour pallier à ce problème sécuritaire, chaque application nécessitant un téléchargement dynamique de code à l'aide de RMI doit installer dans sa JVM un « **RMISecurityManager** ». Celui-ci servira d'organe de contrôle d'accès et base sa politique de sécurité sur un petit fichier de configuration communiqué à la JVM, lors de son lancement. Ce fichier définit un ensemble de « permissions » attribuées au code d'une application. Ces permissions se basent sur la provenance du code et sur la signature éventuelle qui l'accompagne. Comme nous l'avons déjà précisé dans la section 2.4.3, un développeur d'une application cliente RMI sait à l'avance où se situe le RMIregistry (par son URL) et le nom de l'objet remote qu'il souhaite contacter. Il sait également d'où vient le code et qui l'a éventuellement signé. Il peut donc accompagner son application du fichier de configuration sécuritaire approprié à son logiciel.

Dans le cas de Jini, le développeur d'un client ne sait à priori pas quel sera le code téléchargé pour réaliser le service recherché. Le principe étant d'avantage « dynamique », il est bon pour le client « de savoir à qui il a affaire ». Nous constatons que dans le cas de Jini, il y a une double authentification (identification) et autorisation à effectuer. Le service doit savoir qui le demande afin de justifier son exécution et le client doit savoir de qui vient le proxy pour savoir quels droits d'accès

il doit lui donner. Le principe devient symétrique, le client étant d'une certaine façon un petit serveur pour le proxy.

La dernière caractéristique de Jini qui ait une implication sérieuse sur le développement d'un modèle sécuritaire approprié est son indépendance de protocole entre le proxy et le service. En effet, de part cette propriété, un client reçoit obligatoirement un proxy tel une « boîte noire » dont il ignore totalement le fonctionnement interne. Il sait ce que le proxy doit normalement faire, mais il ne sait pas comment il va le faire. Dans un prochain chapitre, nous verrons comment certains travaux tentent de résoudre ce problème.

5.2 Analyse de bas niveau

5.2.1 Les attaques possibles

Comment ferait, d'un point de vue pratique, un ennemi pour attaquer le système ?

Jini est une architecture dite « middleware ». Ceci signifie qu'elle est qualifiée de couche logicielle intercalée entre deux autres. Si nous regardons la pyramide formée par l'empilement de ces couches (Figure 5.1), nous constatons qu'elle est dépendante au niveau sécurité des couches comme : la plate-forme Java 2 (la JVM), l'OS sur lequel tourne la JVM et enfin des services réseaux comme la pile de protocoles réseau utilisés (TCP/IP, par exemple). La plate-forme hardware peut également être considérée comme une source de failles sécuritaires.

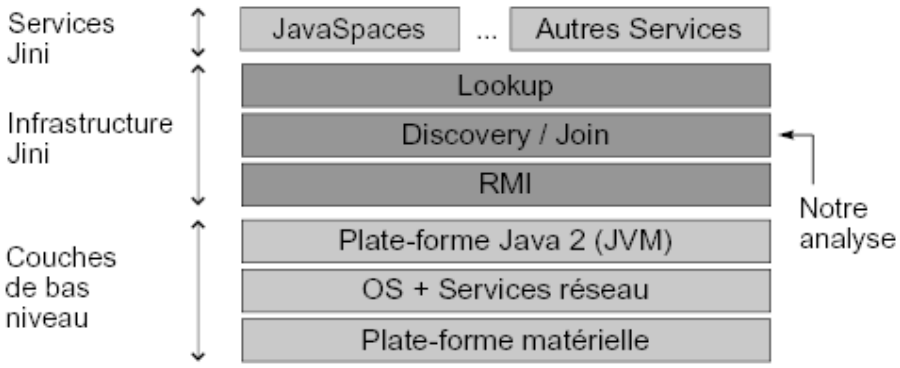

Figure 5.1 : Pyramide des couches logicielles

L'étude sécuritaire de l'ensemble des couches situées en dessous de l'infrastructure Jini font partie de ce que nous appelons « l'analyse de bas niveau ». Néanmoins, évaluer toutes les failles potentielles présentes dans ces couches logicielles revient à

faire un travail titanesque et sort largement du cadre de ce mémoire. Nous soulignons essentiellement le fait que Jini est bâti sur ces couches et que donc, il s'agit également de prendre des mesures de sécurité à ces niveaux là.

Ceci représente le premier type d'attaque « bas niveau » que nous répertorions : un ennemi modifie la couche Jini en passant par les couches inférieures. Par exemple, un petit logiciel installé au-dessus de l'OS modifie « innocemment » les fichiers de base de l'implémentation de Jini.

Il ne faut évidemment pas se tromper, un attaquant ne respectera sûrement pas les règles du jeu ! Et ce, également pour notre deuxième type d'attaque, celui des interactions réseau. En effet, l'ennemi n'utilisera pas forcement l'API Jini originalement développé par Sun Microsystems pour communiquer avec les acteurs d'une interaction Jini (voir section 5.3) mais utilisera plutôt sa propre version faisant office de « simulateur de communication Jini ». Celui-ci cherchera le plus souvent à exploiter une faille dans le processus d'échange de messages comme les protocoles définis par la technologie.

5.2.2 Analyse du protocole de découverte

Nous allons dès lors nous focaliser sur la partie la plus inférieure de l'infrastructure Jini, à proprement parler : le protocole de découverte (Discovery Protocol). Il s'agit du protocole de « bootstrapping » définit par Jini pour permettre à un client d'obtenir le proxy du ou des services lookup de son choix. Les trois variantes de ce protocole ont déjà été énumérées et explicitées dans la section 2.3.2. La question qu'il est intéressant de se poser est la suivante : « Qu'en est-il de la sécurité du protocole de découverte ? »

La première chose à noter est qu'il se déroule à un niveau plus bas, au niveau des communications réseau. L'implémentation du protocole, dans sa spécification actuelle, dépend d'un réseau IP acceptant les paquets de transport UDP « multicast » et les connexions basées sur le protocole de transport TCP. Un ennemi pourra dès lors déjà s'attaquer à ces différents composants.

La disponibilité des connexions

Un programmeur malintentionné cherchera dès lors à congestionner les ports d'un lookup service. Le style d'attaque concerné est celui des attaques de déni de service. Ceci n'est pas une exclusivité de Jini, car toute application réseau est soumise à ce type de risque. Le protocole de découverte de Jini est bien conçu pour ce qu'il doit faire mais ne peut rien contre ce type d'attaque.

L'anonymat

Ensuite, le protocole de découverte pourrait trahir la présence d'un des acteurs sur le réseau alors que celui-ci ne désire être vu que par ses partenaires de communication. Pour arriver à cela, le tout est de savoir ce qui est réellement transmis dans les paquets UDP de découverte. Les seules informations, déjà décrites dans la section 2.3.2, indispensables à son fonctionnement sont : l'adresse IP de la source, l'adresse IP du destinataire (éventuellement multicast), les ports respectifs (origine, destination et de rappel), le ou les groupes visés et une petite information indiquant que nous sommes en présence d'un paquet de découverte Jini. Tout ceci est valable pour les trois types de protocole de découverte spécifiés dans la section 2.3.2. Nous remarquons qu'aucune information strictement confidentielle de l'utilisateur ne semble être transmise, les adresses IP étant un pré-requis à toute communication réseau. Mais, si nous voulons être précis, une adresse IP est déjà un moyen d'identification et donc, de ce point de vue, nous ne pouvons pas conclure que le protocole de découverte garde l'anonymat complet.

Si nous prenons l'exemple du lookup, il ne désire peut-être pas répondre à certains clients et donc être vu par ceux-ci ? Il s'agit dans ce cas que le lookup soit en mesure d'identifier de manière fiable l'origine des paquets multicast envoyés. Mais nous touchons du coup un sérieux problème technique : pour authentifier un individu, il est nécessaire de procéder à une suite d'échanges de messages sous la forme de « questions – réponses ». La simple présentation « d'une carte d'identité » ne suffit pas car dans le monde de l'informatique, ces cartes sont très facilement falsifiables. Si le but de notre lookup est de ne pas être vu par le client, le simple fait d'initier un protocole d'identification trahira sa présence sur le réseau. Nous pouvons raisonner de la même manière pour un client ou un service ne souhaitant pas être vu durant leur recherche de lookup.

Nous n'avons donc pas de solution pour l'anonymat au sens « visibilité des acteurs Jini » durant le processus de découverte. Mais pour ce qui est de la découverte des services, nous passons obligatoirement par le lookup pour les percevoir et obtenir leurs proxies… Dans la section 7.1, Peer Hasselmeyer et al. proposent une solution basée sur cette idée.

Authentification et intégrité des messages échangés

Avant de détailler ce problème, précisons quelques remarques importantes :

(a) Jusqu'à maintenant, nous avons étudié le protocole de découverte sous son aspect réseau. Or celui-ci intervient également à un niveau d'abstraction supérieur. Il est en effet utilisé par le développeur d'applications Jini. Ce dernier

définit essentiellement les actions à entreprendre lorsqu'un proxy de service est découvert. Pour ce faire, il initie un objet Java (le `lookupDiscoveryManager`) chargé de rechercher pour lui les lookups ayant certaines caractéristiques (membre d'un groupe Jini, sa localisation, …). De même, le programmeur doit également définir un objet (le `DiscoveryListener`) qui est à l'écoute des événements (`DiscoveryEvent`) générés par le `lookupDiscoveryManager` lorsque celui-ci a découvert un ou plusieurs proxies de services appropriés. Ce mécanisme respecte le fonctionnement standard de Java. Vous l'aurez deviné, le programmeur reçoit par l'intermédiaire du « `DiscoveryEvent` » le ou les proxies découvert(s). Il est important de souligner que ceux-ci sont des objets Java initialement sérialisés pour la transmission mais qui sont à présent instanciés dans la JVM du client ! (La méthode « constructeur » de l'objet peut déjà contenir du code malicieux)

(b) Le protocole de découverte n'exploite pas la sémantique RMI pour le transfert du proxy du lookup vers le client ! A la place, Jini exploite les facilités de sérialisation du langage Java et sa possibilité de créer des connexions réseau sur base des sockets.

(c) Pour être plus précis, lors du transfert du proxy du lookup, celui-ci n'est pas transmis sous la forme d'un objet Java sérialisé mais sous la forme d'un objet Java « marshallisé » (`java.rmi.MarshalledObject`). Ceux-ci sont des conteneurs d'objets Java sérialisés. Ainsi, cet objet conteneur est bel et bien sérialisé pour sa transmission puis désérialisé automatiquement à son arrivée mais il faut invoquer explicitement une de ses méthodes pour instancier, et donc récupérer, l'objet Java initial qu'il transporte.

Raisonnons à présent sur l'authentification et l'intégrité des messages échangés. Nous l'avons déjà dit dans le cas de l'anonymat, il est difficile d'authentifier un partenaire directement au niveau du protocole de découverte. Les paquets de découverte, comme leur nom l'indique, sont destinés à découvrir d'autres entités. Ils contiennent donc des informations interprétables par tout lookup.

Je pense également qu'il est difficile de protéger leur contenu d'autant plus qu'ils doivent avoir une taille limitée (paquet UDP) ! Il est clair qu'ils peuvent être soumis à des attaques par répétition de paquets ou par modification de leur contenu. Mais n'oublions pas que les paquets de découverte multicast sont normalement destinés à un environnement réseau limité (leur portée est souvent limitée à un LAN). Si nous arrivons à communiquer avec un lookup environnant qui est de confiance, alors celui-ci peut éventuellement joindre un autre lookup de confiance plus éloigné via le protocole de découverte « unicast » et utiliser une connexion sécurisée. L'éloignement

dans un réseau augmente en général le nombre d'entités intermédiaires nécessaires au bon transfert du paquet et donc, augmente par la même occasion le nombre d'attaques potentielles sur ce paquet. Notre client pourrait dès lors accéder à des services plus éloignés. Les paquets UDP de découverte ne sont donc pas vraiment protégeables et garantir leur intégrité semble être une charge supplémentaire mal appropriée.

Cependant, n'oublions pas la deuxième phase du protocole qui consiste à transmettre le proxy du lookup à travers une connexion TCP/IP. Une idée serait d'utiliser une connexion TCP associé à SSL, par exemple, pour assurer l'authentification des parties et l'intégrité du transfert du proxy. Pour ce faire, notre paquet UDP de découverte devrait alors contenir une information supplémentaire précisant le type de connexion obligatoire ou praticable pour l'envoi du proxy. Comme chaque information est placée à un endroit fixe dans le paquet et que chaque information de longueur variable est quand-même délimitée par un entier, nous pouvons éventuellement ajouter cette information en fin de paquet sans perturber le fonctionnement actuel de Jini.

Une autre solution vient de la remarque suivante : « Ce que nous cherchons essentiellement, c'est de protéger l'intégrité du proxy et l'authentification de son origine … plus que la recherche de sa confidentialité ! » Dans ce cas, l'adjonction d'un simple certificat de contenu suffirait à vérifier l'intégrité du proxy et sa provenance. Mais faut-il encore procéder alors à l'authentification de l'émetteur.

Finalement nous constatons que, pour le transfert sécurisé du proxy du lookup au client, nous pouvons utiliser une connexion sécurisée du niveau du réseau (TCP avec SLL) ou bien ajouter un mécanisme de sécurité par-dessus. Dans le premier cas, il y a légère modification du protocole et dans le deuxième également mais au niveau de l'implémentation de Jini. Mais, dans les deux cas, ce qui importe fort, c'est de ne pas instancier le proxy avant qu'il n'ait pu être certifié ! Comme Jini reçoit un objet marshallisé, l'implémentation doit vérifier cette cohérence avant de « libérer » le proxy et le transmettre au client par un « `DiscoveryEvent` ». Pour sa part, le développeur devrait pouvoir préciser à l'API Jini qu'il souhaite utiliser telle ou telle méthode de certification, de n'accepter que les lookups correspondant à certains certificats, etc…

Jini est suffisamment flexible pour pouvoir ajouter ces fonctionnalités tout en facilitant la vie du programmeur et en restant compatible avec les anciennes applications déjà développées. Pour plus d'information sur une solution éventuelle, je vous invite à consulter l'article [7].

Pour l'après processus de découverte, c'est à dire le processus d'enregistrement (*join*) et le processus de recherche de services (*lookup*), nous nous reportons au proxy du lookup car c'est lui, à présent, qui est chargé des communications avec le lookup.

5.3 Analyse de haut niveau : acteurs d'une interaction Jini

Voici, pour rappel, le schéma général des interactions réalisables dans une communauté Jini (voir Figure 5.2). Parmi nos acteurs, nous avons :

- le Lookup Service (LUS ou **L**)
- le Client (**C**)
- le Service (**S**)
- le Proxy (**P**) du service
- la partie Sérialisée du Proxy ($\mathbf{P_S}$)
- la partie Bytecode du Proxy ($\mathbf{P_B}$)
- le serveur de Bytecode (**B**) pour télécharger l'implémentation du proxy

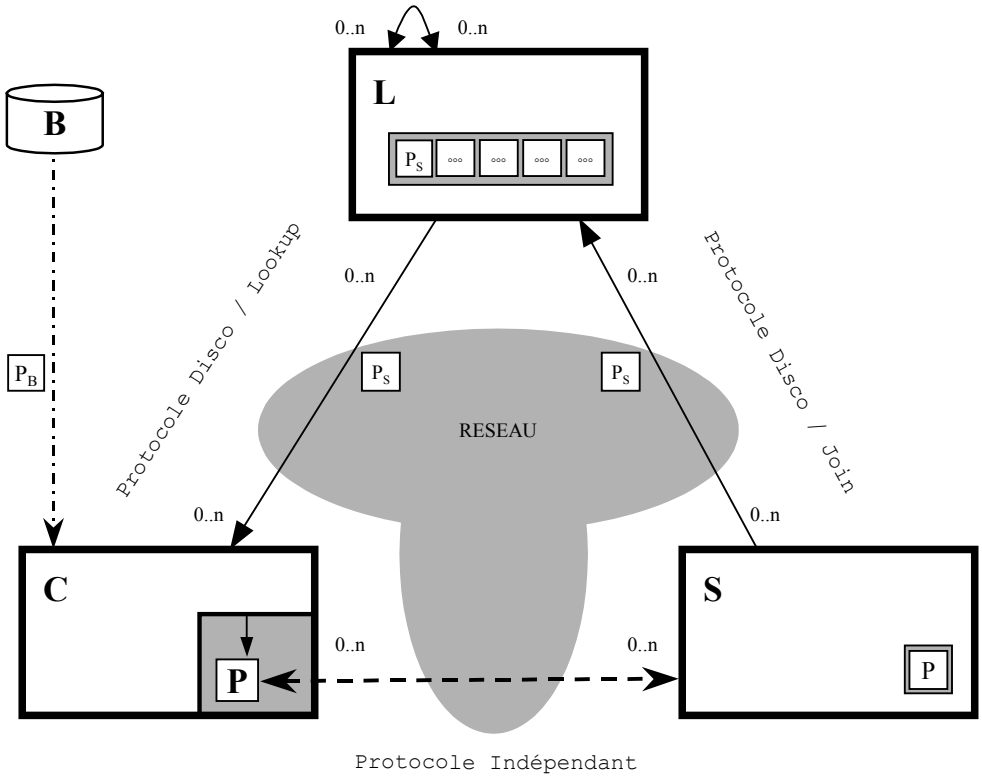

Figure 5.2 : Acteurs d'une interaction Jini

Notre schéma reprend le cas le plus extrême d'un point de vue sécuritaire. C'est à dire, le cas où le client, le service et le service lookup tournent sur des machines différentes et sont séparées par un réseau.

Puisque les échanges d'information peuvent se produire sur un réseau ouvert, comme Internet, n'importe qui peut potentiellement lire et/ou modifier le flux d'informations circulant entre deux acteurs. Nous introduisons dès lors un nouvel acteur, appelé BlackHat (**BH**), qui est capable de s'immiscer dans une communication entre deux autres acteurs Jini.

Il ne faut pas oublier que ces entités informatiques travaillent pour le compte de personnes humaines. Celles-ci peuvent jouer un rôle important sur le comportement de mécanismes sécuritaires futurs. En conséquence, nous les introduisons dès à présent. Nous avons l'utilisateur (**U**) du client. C'est la personne pour laquelle le client s'exécute. Nous avons aussi le fournisseur (**F**) du service. C'est la personne pour laquelle le service s'exécute. Nous pourrions faire de même pour le service lookup mais comme déjà précisé, le lookup est un service comme les autres, et si ces deux acteurs jouent un rôle important dans une solution sécuritaire quelconque, alors le service lookup pourrait implicitement hériter de cette dernière.

Dans le cas d'une analyse sécuritaire, il est bon de savoir ce que nous désirons protéger. Il est clair maintenant qu'au niveau d'une interaction Jini les acteurs concernés sont : C, L, S, U, F et B.

C, L et S sont des processus au sens informatique du terme. Il est donc nécessaire de les protéger des attaques qui tenteraient à les rendre inopérants. Ce genre d'attaque cherche soit à corrompre les données nécessaires à leur bon fonctionnement (intégrité des données), soit à les surcharger de travail (déni de service).

Mais les acteurs les plus touchés au bout de la chaîne sont les utilisateurs U et F. Dans leurs cas, il s'agit essentiellement de protéger leurs données, confidentielles ou non (intégrité et confidentialité des données).

Pour effectuer notre analyse, plaçons-nous dans la peau de chaque acteur participant au réseau Jini. Pour ce faire, nous allons imaginer que cet acteur est malicieux et qu'il cherche donc à nuire aux autres acteurs de la relation. Nous introduisons la notation « * » pour désigner un acteur malicieux.

5.4 Cas du Proxy malicieux (P)*

Cherchons dans un premier temps, les différentes manières d'introduire P* dans la relation.

(1) BH intercepte le proxy sérialisé P_S que le service S désire enregistrer dans le service lookup L. Durant le transfert, il le remplace par sa propre version.

(2) BH intercepte le ou les P_S transmis par L à C lorsque ce dernier a décidé d'effectuer une recherche de services. A nouveau, il le(s) remplace par sa propre variante.

(3) BH intercepte P_B lorsqu'il est installé sur le serveur de fichiers B et le remplace par sa propre version.

(4) BH intercepte P_B lorsqu'il est automatiquement téléchargé par le client C depuis B.

(5) BH attaque le serveur B et installe son propre P_B*.

(6) L peut être lui-même malhonnête et distribuer des P_S* qui sont associés à des P_B*.

(7) Un S* malicieux enregistre son P_S* malicieux dans un L.

(8) Le client C* malicieux a fabriqué lui-même un mauvais P* à partir d'un bon P.

Maintenant que notre P* malicieux est activé dans la JVM du client, cherchons les différentes menaces qu'il représente envers les autres acteurs.

5.4.1 P* attaque le client C

(1) Le proxy malicieux lit des informations confidentielles, puis les transmet à son service malicieux (confidentialité des données du client).

(2) Le proxy malicieux tente d'écrire sur une ressource de C (intégrité des données du client).

(3) Le proxy surcharge des ressources du client de façon à le ralentir ou à le bloquer (déni de service du client).

5.4.2 P* attaque un service S

(1) Le proxy se fait passer pour un proxy de S, récolte des informations confidentielles, puis les transmet au client malicieux (confidentialité des données du service).

(2) Le proxy envoie des informations incorrectes à S (intégrité des données du service).

(3) Le proxy inonde S de requêtes de façon à le rendre inopérant (déni de service du service).

5.5 Cas du Client malicieux (C)*

5.5.1 C* attaque un L

L étant un service, nous pouvons considérer que C* l'attaque à l'aide du proxy modifié de ce LUS. Nous nous reportons donc à ce cas. Cependant, l'interaction « client-lookup » ne se limite pas seulement à celle du proxy. En effet, avant ce stade, le client et le lookup utilisent le protocole de découverte pour s'apercevoir. Nous avons déjà étudié son aspect sécuritaire dans une section précédente (section 5.2.2).

5.5.2 C* attaque un P

Rendre un proxy inopérant ne présente pas vraiment un problème car le proxy travaille pour le client dans la machine de ce dernier. Le défi serait plutôt de modifier un proxy pour l'utiliser contre un autre acteur. Nous revenons à un cas précédemment analysé (section 5.4). Cependant, nous pouvons imaginer qu'un proxy viendrait avec des informations confidentielles. Une attaque éventuelle serait d'essayer d'obtenir ces informations (confidentialité).

5.5.3 C* attaque un S

Nous considérons que l'attaque se fait à l'aide d'un P*. Nous nous rapportons donc à l'étude de ce dernier.

5.6 Cas du LUS malicieux (L)*

5.6.1 L* attaque un C

(1) Comme déjà vu pour le cas du client, L* pourrait attaquer des clients au niveau du protocole de découverte. Ce n'est que le cas symétrique.

(2) L* peut transmettre des faux proxies à ses clients. (voir P*)

(3) De façon générale, C communique avec L à l'aide du proxy de L. L* peut donc attaquer C à l'aide de son proxy. Nous nous référons au cas d'un proxy malicieux attaquant un client.

(4) L* étant le centre névralgique de toute recherche de services, il est en mesure de répertorier les besoins de ses clients en service et de revendre les informations ainsi récoltées. Nous touchons à la confidentialité des individus (vie privée).

5.6.2 L* attaque un S

(1) Attaque par le protocole de découverte.

(2) Attaque de déni de service.

(3) L* attaque S, qui pour lui est un client, à l'aide de son proxy. Nous nous référons au cas d'un proxy malicieux attaquant un client.

(4) L* peut informer des parties intéressées par l'existence d'un certain service chez le fournisseur concurrent.

5.6.3 L* attaque un P

L* peut, comme un client malicieux, modifier les proxies de ses services afin de les transmettre à des clients.

5.7 Cas du Service malicieux (S)*

5.7.1 S* attaque un L

(1) S* transmet un P* à L. Nous nous référons au cas de P*.

(2) S* comme un C* tente de rendre le L inopérant par une attaque de déni de service.

5.7.2 S* attaque un C

S* interagit avec C par l'intermédiaire du proxy de S* que nous considérons comme malicieux car complice. Cette constatation nous ramène à évaluer le champ d'action du proxy malicieux car c'est lui qui est en contact direct avec le client.

5.8 Cas de BlackHat (BH)

En plus de nos acteurs principaux, BH est un acteur qui a la possibilité d'agir sur chaque liaison de communication. Il peut donc avoir les influences suivantes :

(1) Perturber le protocole de découverte : en empêchant les bons échanges de paquets IP ou en transférant son proxy malicieux à la place de celui de L impliqué dans le protocole. Mais alors, nous pouvons considérer que BH devient un L* pour S ou C (intégrité des messages, confidentialité des messages).

(2) Perturber le processus de recherche : en injectant son propre proxy malicieux lors de la réponse de L à C (intégrité des messages, confidentialité des messages).

(3) Perturber le processus d'enregistrement : en injectant son propre proxy malicieux lors du transfert du proxy de S à L (intégrité des messages, confidentialité des messages).

(4) Perturber les échanges d'information entre un proxy P et son service S (intégrité des messages, confidentialité des messages).

(5) Perturber le téléchargement du code du proxy à partir du serveur B : en transmettant sa propre implémentation par exemple ! (intégrité des messages)

(6) Perturber n'importe quelle communication réseau (déni de service).

5.9 Menaces externes à nos acteurs : « Mascarade »

Nous avons supposé durant notre analyse précédente que les ennemis du système étaient soit des acteurs malicieux, soit un BlackHat désirant perturber les communications. Mais parmi les attaques bien connues des systèmes distribués, celle de la « **mascarade** » n'a pas encore été soulignée jusqu'à présent. De par sa définition, cette attaque consiste à usurper l'identité d'une des deux parties en cours de communication. BlackHat étant notre « usurpateur d'identité » dans ce cas-ci, il continuera à dialoguer avec une des deux entités laissant l'autre seule face à une communication réseau rompue. Vous l'aurez tout de suite compris, ceci n'est qu'une autre façon d'introduire une de nos entités malicieuses précédemment définies dans la relation. Nous pouvons dès lors nous ramener aux cas précédents.

5.10 Conclusions : Exigences nécessaires en sécurité

Comme nous pouvons le constater, la plupart des attaques nous amènent à considérer le proxy comme une menace sérieuse. De tous ces cas de figure, il s'agit de mettre au point des mécanismes de protection pour les points suivants.

5.10.1 Protection du client et de ses ressources

Nous devons penser ici à contrôler l'accès à ses ressources. Celles-ci sont du type : fichiers, connexion réseau, mémoire, CPU, … N'importe quel proxy ne doit pas pouvoir accéder à n'importe quelle ressource du client. Même un proxy que nous pourrions par un moyen ou un autre considérer comme fiable ne doit pas forcément avoir les pleins pouvoirs.

Confidentialité

Concernant la protection des ressources du client, nous pensons à celles capables de stocker de l'information : disque dur, mémoire, … Il s'agit donc de contrôler l'accès en lecture. Du côté de la protection des utilisateurs, il doit pouvoir garder un certain anonymat sur le réseau Jini. Il ne souhaite pas forcément que tous les services puissent établir sa présence.

Intégrité

Nous pouvons faire les mêmes observations que pour la confidentialité. Cependant le contrôle d'accès se fait en écriture sur la ressource.

Disponibilité

Nous pensons plus particulièrement à des ressources critiques comme le processeur et la mémoire. Mais tous les autres types sont également concernés. Il s'agirait de mettre au point un mécanisme capable de contrôler l'utilisation qu'un processus fait d'une ressource.

5.10.2 Protection des messages échangés

Confidentialité

Aucune personne étrangère à la relation ne doit être capable de lire ou bien de comprendre la signification des messages échangés entre nos différents acteurs : que ce soit des messages nécessaires au bon fonctionnement de Jini ou des messages contenant des informations confidentielles des utilisateurs. Le chiffrement est la

solution idéale contenu du fait que l'information peut transiter sur un réseau ouvert. Mais si nous avons la possibilité de faire passer celle-ci par un chemin fiable, alors nous pouvons nous en passer.

Intégrité

Personne ne doit être capable de modifier le flux d'informations sans que les parties concernées ne s'en rendent compte. A nouveau, le mécanisme de signature digitale est une solution de premier choix.

5.10.3 Protection du service et de ses ressources

Confidentialité

Du point de vue de la philosophie Jini, chaque service, aux yeux du client, peut être considéré comme une ressource utilisée temporairement par celui-ci. Ces services seraient, de ce point de vue, les ressources appartenant à notre fournisseur de services F. Si nous protégeons l'accès aux ressources d'un client, ou plus exactement « de la machine de l'utilisateur U du client », il serait également bon de protéger l'accès aux ressources du fournisseur F. Il faudrait un mécanisme capable non seulement d'empêcher des clients, n'ayant pas droit à l'utilisation d'un service, de se voir refuser l'accès à ce service mais également de pouvoir contrôler la visibilité qu'ils ont de ces services. Ceci n'est clairement pas le cas de la version actuelle de Jini qui offre l'accès à tous les services pour tout le monde. Le contrôle de visibilité ne doit pas se limiter à la présence ou non du service sous les yeux du client mais doit également contrôler la visibilité et l'accessibilité de ses sous-fonctionnalités représentées ici par les méthodes de l'interface du proxy. Nous étendons le mécanisme de contrôle d'accès aux réseaux de services.

Intégrité

Un service n'exécute normalement pas de code mobile au sein de sa JVM. Sauf, bien évidemment, si le programmeur lui rajoute cette fonctionnalité ou s'il est le client d'un autre service. Les seules interactions qu'il a avec son client passent par son proxy. Or, si ce dernier est modifié, il pourrait changer l'ordre d'exécution du protocole de communication et, par la même occasion, provoquer des problèmes d'intégrité de données du côté du service. Mais malheureusement, ceci relève plus de la tâche du développeur du service que de mécanismes de sécurité offerts par Jini. Par exemple : nous possédons un service capable de lire des fichiers sur un système distant. Le protocole prévoit trois étapes : l'ouverture, la lecture et la fermeture du

fichier. Si un proxy malicieux saute la dernière étape, les fichiers du serveur resteront ouverts. L'intégrité des fichiers aura été corrompue.

Disponibilité

Un service doit rester disponible, même en cas de forte demande. C'est ce vers quoi Jini tend dans sa version originale. En effet, un service peut être redondant sur le réseau ou redémarrer spontanément après un plantage de sa machine hôte grâce au mécanisme de persistance des objets. L'indisponibilité d'un service peut simplement venir d'une panne matérielle ou logicielle. Mais dans le cas d'un acte malveillant de déni de service, il serait avantageux que Jini offre aux développeurs un mécanisme permettant un contrôle sur ce que les clients peuvent faire (nombre de connexions par exemple) et sur l'identité de ces clients. Il serait dès lors envisageable de prendre des mesures à un niveau plus élevé pour contrer ce genre de menaces.

5.10.4 Problèmes sécuritaires spécifiques à Jini

La principale caractéristique ayant un impact sur la sécurité et qui distingue Jini des autres technologies offrant une infrastructure pour la communication d'objets distribués, est sa tendance à exploiter intensivement le téléchargement dynamique de code offert par le langage Java : « le code mobile ». Cette particularité oblige Jini à offrir aux utilisateurs des mécanismes capables de résoudre principalement trois problèmes.

Le contrôle d'accès mutuel (aux ressources)

Dans une sécurité réseau traditionnelle, c'est le serveur qui offre d'une certaine façon ses ressources aux clients. Il lui est donc capital de pouvoir, dans un premier temps, identifier son client puis, sur base de cette identité, lui offrir des accès différents.

Dans le cas de Jini, c'est différent. Le client maintenant possède une partie du code du serveur qui utilisera les ressources du client. Il s'agit donc d'effectuer une identification et autorisation mutuelle.

Etablir la confiance du proxy « opaque »

Ayant reçu un proxy appartenant selon nos espérances à un certain fournisseur de services, comment pouvons-nous être certain que ce proxy ne fera pas d'actions à notre insu ? Il est évidemment très difficile de vouloir contrôler tous les faits et gestes du proxy durant son exécution. Ceci va même à l'encontre de la notion d'indépendance de protocole pour laquelle il est créé. La seule solution viable est celle de la confiance. Cette question requiert en général une réponse « binaire ».

Avons-nous confiance en ce proxy ? : « Oui/Non ». Mais il y a peut-être moyen de nuancer : « Oui, à 63% ». Le proxy et le service formant ensemble le service offert par un fournisseur, si nous avons confiance en ce fournisseur, alors il est concevable d'avoir confiance en ses services. De même, si nous avons confiance en ses services, il est logique d'avoir confiance en son proxy. La question se transforme donc en « Comment prouver que le proxy est bien celui du service recherché ? ».

Vérifier l'intégrité des objets reçus (le proxy)

Il s'agit d'établir l'intégrité totale de l'objet reçu. Il ne faut pas qu'il ait été altéré durant son transfert. Il faut aussi pouvoir vérifier qu'il a bien été émis par tel fournisseur. L'intégrité totale regroupe l'intégrité de l'objet Java sérialisé qui représente le proxy ainsi que son code téléchargé à partir du codebase présent dans l'objet Java sérialisé.

Pour résumer…

Nous résumons ici les caractéristiques principales de Jini ayant un impact direct sur la sécurité :

- **Code mobile** : D'où vient le proxy, de qui vient-il, est-il bien celui que nous pensons, que va-t-il faire sur nos ressources, que fait-il avec nos informations ?
- **Indépendance de protocole** : Elle est la cause de la vue du proxy comme une boîte noire.
- **Libre découverte** : Chaque client peut découvrir et accéder à n'importe quel service sans contrôle préalable.

5.11 Liste des exigences en sécurité et éléments de solution

Dans cette partie, nous déterminons un ensemble d'exigences en sécurité nécessaires aux applications Jini. Ces exigences sont utilisées comme **référence pour comparer** les différentes architectures présentées dans un chapitre suivant. Pour chacune de ces exigences, nous ajoutons également un ou plusieurs élément(s) de solution potentiel(s) ainsi que les difficultés liées à l'exploitation de ces solutions. L'ensemble des solutions émises nécessite l'utilisation de techniques cryptographiques (comme la cryptographie à clé publique et à clé secrète) et nécessite également l'architecture sécuritaire offerte par la plate-forme Java 2 (entre autres pour les contrôles d'accès). Nous verrons que les différentes architectures proposées tentent le plus souvent de

résoudre un sous-ensemble de ces exigences. De plus, certaines exigences ne sont toujours pas résolues…

Remarques :

- Ces exigences en sécurité ne doivent bien entendu pas toutes être forcément remplies par une architecture donnée car leur nécessité dépend essentiellement du domaine d'application envisagé.
- Afin de raccourcir les exigences, nous utilisons les notations déjà définies dans la section 5.3.

Exigences d'authentification (identification)

(E1) C aimerait être certain que P vient de S et que S est un service de F.

(E2) P aimerait être certain qu'il parle à son S.

(E3) S aimerait être certain qu'il parle à son P.

(E4) P aimerait être certain qu'il est exécuté par un bon C.

(E5) S aimerait être certain que c'est C qui utilise P.

(E6) C aimerait être certain qu'il parle à S.

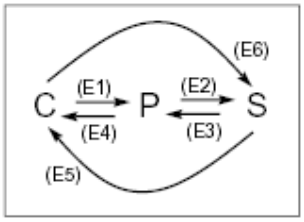

Figure 5.3 : Relation « X authentifie Y »

Remarque :

La Figure 5.3, résume les six premières exigences. La flèche représente la relation suivante : « l'acteur X authentifie l'acteur Y ».

Précédemment nous avions constaté que L était un service comme les autres et que par conséquent, si nous arrivions à mettre en place des mécanismes sécuritaires pour les communications « client-service », alors il en serait probablement de même pour

le service lookup. Cependant, nous avons constaté dans la section 5.2.2 que le proxy du lookup s'obtient par le processus de découverte et que l'authentification et le contrôle de l'intégrité de ce proxy doit être faite par Jini avant que le proxy soit instancié. Nous pouvons y arriver en modifiant légèrement l'implémentation de Jini et son protocole de découverte (tout en gardant une compatibilité descendante). Ceci vient du fait que ce mécanisme est figé et implémenté par du code « certifié » puisqu'il s'agit de celui fournit par Sun Microsystems et qui est installé sur la machine du client (donc local).

Dans le cas des proxies des autres services, nous les obtenons par l'intermédiaire du proxy du lookup ! Le code du proxy a été téléchargé dynamiquement depuis le réseau. Si nous le considérons comme fiable, il n'implémente pas forcément un mécanisme capable de vérifier l'authenticité et l'intégrité des proxies des autres services. En particulier, il ne procède pas forcément à cette vérification avant que l'objet proxy du service recherché ne soit instancié.

Exigences de confidentialité et d'intégrité

(E7) C aimerait que P utilise un moyen de communication sécurisé avec S pour la transmission de ses données confidentielles (évite le vol d'information par une tierce personne).

(E8) S aimerait que les informations confidentielles qu'il donne à C par l'intermédiaire de P ne soient pas interceptées par une tierce personne.

(E9) S voudrait que son P ne soit pas distribué à n'importe qui.

Exigences de contrôle d'accès (autorisation)

(E10) C aimerait que P ne puisse pas accéder à ses données confidentielles.

(E11) C aimerait que P n'envoie pas les données confidentielles à S.

(E12) L ne souhaite recevoir que les P venant de certains S.

(E13) C aimerait contrôler de façon générale l'accès qu'a P sur ses ressources.

(E14) S aimerait limiter les capacités de son P (et donc de ses services offerts) en fonction de C ou bien de U : contrôle d'accès des méthodes de P ou contrôle de leur comportement.

(E15) C aimerait contrôler la quantité d'utilisation de ses ressources.

(E16) S aimerait contrôler la quantité d'utilisation de ses ressources.

Exigences techniques

(E17) Il faut rester compatible avec les anciennes applications Jini non-sécurisées (client, service et lookup).

(E18) Il faut offrir la sécurité de façon transparente au développeur (il n'est pas forcément un spécialiste de la sécurité).

(E19) L'architecture doit être facile à utiliser.

(E20) Les performances constatées et les ressources nécessaires doivent être appropriées au contexte d'utilisation.

Exigences philosophiques

(E21) Il faut préserver l'indépendance de protocole (Proxy-Service).

(E22) Il faut préserver la libre découverte des services.

(E23) Il faut préserver « l'auto-réparation » offerte par Jini.

Idées de solution et difficultés liées

Dans cette sous-section, nous donnons rapidement des idées de solutions potentielles pour chacune des exigences précédentes. Nous nous arrêtons cependant à l'exigence (E16) car les suivantes, appartenant au groupe des exigences techniques et philosophiques, nécessitent une architecture complète afin d'y répondre de façon détaillée. Ces dernières exigences seront tout particulièrement utilisées pour comparer les deux architectures proposées dans le chapitre 7.

N°Ex.	*Elément(s) de solution*	*Difficulté(s) liée(s)*
(E1)	C vérifie l'intégrité du P reçu en vérifiant la signature de P_B et de P_S. Ces signatures ont été réalisées par S. C utilise la K_P de S pour la vérification.	- La K_P de S doit être obtenu d'une façon ou d'une autre (par un CA, par une chaîne de confiance venue avec P ou arrivée par le lookup).

N°Ex.	*Elément(s) de solution*	*Difficulté(s) liée(s)*
(E2)	Si P obtient un moyen de communication avec S alors il peut implémenter le protocole d'authentification de son choix.	- Il faut donner à P un accès aux ressources de C. Lesquelles choisir ? Sur quelle base ? - P doit également venir avec une information permettant de reconnaître son S. Comme P est exécuté par C, P ne peut pas vérifier par lui-même qu'il n'a pas été modifié. Il dépend donc de C et de ce point de vue, authentifier S par C a plus de sens que S par P.
(E3)	Impossible : P ne peut transporter avec lui aucune information secrète (comme une K_C ou une K_S) susceptible de l'authentifier auprès de S : ce secret peut être volé par une personne malveillante. La seule personne qui peut certifier la « bonne forme » du proxy, c'est celui qui l'utilise (C).	- Il faut que S puisse préalablement authentifier C. - Il faut ensuite que S ait confiance en C. (faut-il un mécanisme supplémentaire pour déterminer la confiance ?)
(E4)	Difficilement applicable : P est fortement lié à C puisqu'il est exécuté par celui-ci. P ne peut pas non plus garder secret des informations lui permettant d'identifier un bon C car P a pu être modifié sans qu'il puisse s'en rendre compte. Il est donc plus judicieux d'identifier C par S car c'est surtout lui au bout du compte qui a besoin d'authentifier C afin de protéger ses ressources (cf. exigence (E5)).	

N°Ex.	*Elément(s) de solution*	*Difficulté(s) liée(s)*
(E5)	C doit s'authentifier auprès de S via un protocole d'authentification. Celui-ci nécessite en général la K_C de C. Le protocole est implémenté à un : niveau inférieur : via SSL, TLS, … niveau supérieur : via P ou un API standardisé.	- S doit connaître la K_P de C. - Peut-on vraiment authentifier à travers P ? (oui, si l'identification de C est laissée aux soins de S alors S a la responsabilité de créer un P capable d'authentifier un C) - Dans ce cas, P doit pouvoir demander à C une preuve de son identité (nécessite un mécanisme standard).
(E6)	Si nous avons confiance en S et que nous considérons qu'un S de confiance crée obligatoirement un P de confiance alors résoudre (E1) implique (E6). Sinon nous devons passer par un protocole standard figé.	- Un protocole standard figé, va à l'encontre de l'indépendance de protocole de Jini.
(E7)	Si P a reçu de C des informations confidentielles qu'il doit transmettre à S et qu'il est autorisé à le faire par C, nous constatons deux solutions :	- Hypothèse : S est de confiance car sinon il peut diffuser l'information après sa réception.

N°Ex.	*Elément(s) de solution*	*Difficulté(s) liée(s)*
	- Si l'exigence (E1) est satisfaite et que C a confiance en S et en ses P alors C a une confiance totale envers le proxy qui met tout en œuvre pour assurer le transport sécurisé de l'information confidentielle.	- Si P compte chiffrer l'information, il a besoin d'une clé secrète commune avec S mais P ne peut la transporter avec lui. - P doit donc soit utiliser un protocole d'échange de clés secrètes, soit utiliser une entité externe qui distribue des clés secrètes. Dans les deux cas, P doit posséder un couple de clés asymétriques (K_P/K_C) pour obtenir K_S. Seul le client C peut le lui fournir.
	- C impose à P ses moyens de communication sécurisés standards en fonction de ce que le proxy est censé réaliser.	- Imposer un moyen de communication au proxy touche à l'indépendance de protocole de Jini. - Comment se mettre d'accord sur ce qui doit être proposé comme moyen de communication et ce qui doit être accepté ? (protocole de négociation entre C et P) - Ceci nécessite que C et S soient en accord sur ce qu'est un moyen de communication sécurisé standard.
(E8)	S doit créer lui-même un P capable d'établir les connexions sécurisées nécessaires.	- Hypothèse : C est de confiance car sinon il peut diffuser l'information après sa réception. - Problèmes cryptographiques identiques à l'exigence (E7).

N°Ex.	*Elément(s) de solution*	*Difficulté(s) liée(s)*
(E9)	L est la seule entité capable de contrôler la visibilité et la distribution des services à moins que les P soient totalement chiffrés par S et que seuls les C à qui ces P sont destinés soient en mesure de les déchiffrer. Mais cela ne résout pas le problème de visibilité car pour obtenir un proxy, il faut le décrire et si C l'obtient sur base de sa description, cela signifie qu'il existe et qu'il est donc visible. Une solution serait donc d'associer les services par groupe. Chaque C possèderait des droits de recherche (visibilité) et de téléchargement de P appartenant au groupe donné. De même les services pourraient enregistrer leurs P dans un L sur base de leur droit « d'enregistrement ».	- On doit d'une certaine façon créer un réseau de L sécurisés. - Les C doivent être en mesure de prouver aux L qu'ils ont le droit de voir et d'obtenir les S demandés. (voir capabilities) - Chaque L doit-il connaître tous les C pour tous les S ? - Qui administre ces droits et où le fait-il ? - Le transfert des droits d'accès des clients vers L modifie-t-il l'interface standard que doit implémenter un proxy de lookup ?
(E10)	Il faut utiliser les mécanismes de contrôle d'accès de la plate-forme Java 2. Les accès aux ressources sont limités pour tous les P sauf pour ceux de confiance à qui nous pouvons donner plus de droits.	- Comment savoir quels droits d'accès donner pour chaque proxy reçu ?
(E11)	Si P n'a pas les informations confidentielles et s'il ne doit pas les posséder alors cf. exigence (E10). Si P les obtient, il faut contrôler et filtrer tout ce qu'il envoie à S.	- Contrôler le flux d'information émis par P en direction de S est une tâche titanesque voir impossible (ne fut ce que par la possibilité d'envoyer de l'information par canal subliminal).
(E12)	L et S doivent chacun posséder une paire de K_P/K_C. L doit savoir quel(s) service(s) il accepte d'héberger. Il doit donc connaître la clé publique de chacun de ces services.	- L doit être facile à administrer et avoir une sorte de GUI. - Comment L obtient-il ces K_P ?

N°Ex.	*Elément(s) de solution*	*Difficulté(s) liée(s)*
(E13)	Ceci est le cas général de l'exigence (E10).	- cf. (E10).
(E14)	C doit d'abord être identifié par S (cf. exigence (E5)). Ensuite, pour le contrôle de méthode : - Soit C ne voit qu'une interface limitée. - Soit chaque méthode de P peut jeter une exception si le serveur ne veut pas exécuter les opérations correspondantes (suite à l'authentification préalable). pour le contrôle du comportement : - Soit le comportement change du côté du proxy. - Soit le comportement change au niveau du service exécuté sur la machine du serveur.	- Limiter l'interface est impossible car P peut toujours être analysé et modifié. - Quid d'un mauvais P qui s'authentifie normalement puis ferait des opérations illégales par la suite ? - Si P doit modifier son comportement en fonction de C, alors il reçoit l'ordre depuis S. En effet, c'est lui qui authentifie C à moins que C puisse lui fournir une preuve de son authenticité. Mais dans les deux cas, P peut toujours être analysé et modifié par un C*. Ce qui signifie que ce C* peut profiter du travail effectué par P même s'il n'en a pas la permission.
(E15)	Rien n'est inclus, à l'heure actuelle, dans la plate-forme Java 2 pour contrôler cela. Mais nous pourrions créer une « thread » comptabilisant le temps d'utilisation d'une ressource.	- Créer une thread pose un problème d'efficacité : elle est écrite en Java et est donc interprétée par la JVM. Il serait plus efficace que ce mécanisme fasse intégralement partie de la plate-forme.
(E16)	Cf. exigence (E15).	- Cf. exigence (E15).

Chapitre 6

Jini, quelques exemples d'utilisation

Plusieurs exemples concrets nous aident à mieux comprendre l'utilisation de Jini dans le monde actuel et futur. Ces exemples illustrent également les problèmes sécuritaires qui en découlent.

6.1 Les réseaux Jini pour la domotique

6.1.1 Quelques exemples

Chaque appareil se trouvant dans un domicile est, à l'égard de ses occupants, une entité capable de fournir un certain service. C'est dans cette perspective que chaque équipement est comparable à un service Jini.

Imaginons un domicile quelconque. Parmi les équipements électroniques traditionnels, nous trouvons : un réfrigérateur, une cafetière, un magnétoscope, un thermostat, des lampes, un ordinateur personnel, un PDA avec communication sans fil… Evidemment, il s'agit, à l'heure actuelle, d'un domicile de luxe. Imaginons maintenant que chacun de ces appareils soit considéré comme Jini-enabled, que notre magnétoscope soit capable de coder un signal vidéo en un flux numérique, que notre ordinateur devienne un service lookup pour l'ensemble des équipements et qu'il possède un service de stockage d'information sur son disque dur. Nous pourrions également contrôler tous les appareils grâce au PDA mobile qui serait un client Jini. L'adjonction d'un nouvel appareil Jini-enabled dans la communauté serait automatiquement prise en compte. Notre vidéo pourrait utiliser le service de stockage

du PC pour sauvegarder son flux vidéo. La cafetière pourrait être programmée depuis le PDA. Si nous ajoutons également une connexion à Internet, la possibilité de contrôle de l'utilisateur dépasserait les frontières de son domicile. De même, notre vidéo serait à même de dresser un rapport lorsqu'elle détecte une panne matérielle et de l'envoyer automatiquement au service après-vente du constructeur.

6.1.2 Contraintes sécuritaires

Les contraintes sécuritaires propres à ce type de réseau touchent essentiellement les problèmes d'authentification. En effet, il ne s'agit pas que les enfants du domicile puissent modifier en toute liberté la température de la maison ou qu'ils puissent accéder aux chaînes télévisées interdites… Le réseau étant assez limité en envergure et plus ou moins certifié, le chiffrement des échanges de messages semble moins critique. Cependant, avec une ouverture sur un réseau extérieur, la confidentialité et l'intégrité des messages doivent également être prises en compte.

6.2 Les réseaux ad hoc (sans fil)

Les réseaux de mobiles sans fil, peuvent être répertoriés en deux classes distinctes : les réseaux avec infrastructure qui utilisent généralement le modèle de la communication cellulaire (comme le système GSM), et les réseaux sans infrastructure dits « réseaux ad hoc ».

Ces derniers peuvent être définis comme une collection d'entités mobiles interconnectées par une technologie sans fil formant un réseau temporaire sans l'aide de toute administration ou de tout support fixe. Aucune supposition ou limitation n'est faite quant à la taille du réseau ainsi créé.

6.2.1 Quelques exemples

Les applications des réseaux ad hoc sont nombreuses : domaine militaire, opérations de secours, missions d'exploration, …

6.2.2 Contraintes sécuritaires

Dans ce genre d'environnement, il est très difficile de mettre au point des stratégies sécuritaires du moins car les mécanismes standards d'authentification se basent sur la possession préalable d'informations sur les entités environnantes. Dans le cas d'un réseau militaire, les contraintes en sécurité sont de toute évidence très fortes.

Remarquons que de par leur structure hiérarchique, et leur dissociation du monde extérieur (les civiles), les militaires ont la possibilité d'équiper leurs appareils avec des moyens de reconnaissance propres : un appareil militaire n'acceptera une communication qu'avec un autre appareil militaire via des codes de reconnaissance.

6.3 Les réseaux d'appareils médicaux

6.3.1 Quelques exemples

L'ensemble des équipements électroniques habituels d'un hôpital pourrait à son tour intégrer le monde des réseaux Jini.

Imaginons simplement un appareil portatif pour le médecin. Celui-ci lui afficherait automatiquement les informations du patient qu'il est en train d'ausculter. Cet appareil étant Jini-enabled, il pourrait se connecter automatiquement au service lookup présent dans la chambre du patient (ce dernier étant éventuellement relié à d'autres lookups). A l'autre bout du réseau, nous avons le logiciel d'enregistrement des patients utilisé entre autres pour la facturation. Celui-ci pourrait être relié au réseau Jini par l'intermédiaire d'un service le représentant et jouant donc le rôle d'interface.

Dans un hôpital, comme dans d'autres lieux de travail, la maintenance des équipements est un sérieux problème pratique. Jini est naturellement en mesure d'alléger fortement cette tâche car le proxy utilisé par un appareil pour accéder à un service est téléchargé dynamiquement et peut donc varier facilement au fil du temps. De plus, le proxy étant un composant actif, il est en mesure de mettre à jour les paramètres des équipements qui l'auraient téléchargé.

6.3.2 Contraintes sécuritaires

Les contraintes sont évidemment très fortes car la confidentialité et l'intégrité des informations concernant les patients ne doivent pas être ébranlées même pour un transfert sur le réseau de l'hôpital. De même, si les appareils médicaux deviennent des clients Jini (et/ou service), ils sont en mesure de télécharger des proxies … Il ne s'agit évidemment pas que ces proxies soient malicieux et rendent l'appareil inopérant. Il serait très regrettable que le cardiogramme et le défibrillateur soient hors-service durant une urgence.

6.4 Les réseaux d'entreprise

Les réseaux d'entreprise, du point de vue de leurs moyens de communication, sont considérés comme semi-ouverts. En effet, même si les applications utilisées par ses employés se trouvent sur un réseau local isolé, il n'est pas rare que l'entreprise offre un accès à Internet pour des raisons commerciales évidentes. Les protections dans de telles circonstances viseront à séparer l'ensemble du parc informatique du monde extérieur par des mécanismes sécuritaires placés au niveau des réseaux : firewalls, VPN, IPsec, …

6.4.1 Quelques exemples

Imaginons une imprimante avancée d'une entreprise connectée au réseau local de celle-ci. Aux yeux des utilisateurs, elle est perçue comme un service Jini. Or, le pilote nécessaire à toute application bureautique pour envoyer un document à un périphérique d'impression n'est pas forcément installé sur la machine du client. Grâce à Jini, nous recevons le proxy Jini de l'imprimante qui sera utilisé pour le transfert du document.

Imaginons un autre exemple basé ici sur des imprimantes traditionnelles exploitant le port parallèle de la machine du client. A nouveau, notre client ne possède pas le pilote. Par chance, il existe sur son réseau Jini un service distributeur de pilotes d'imprimantes. Nous recevons un premier proxy qui déterminera le modèle de notre imprimante puis téléchargera un autre proxy capable de contrôler notre équipement. Pour ce faire, le dernier proxy utilisera les communications du port parallèle. Nous constatons par cet exemple, que les communications effectuées par un proxy ne se limitent pas au réseau et à son protocole mais peuvent exploiter n'importe quel moyen d'échange d'informations.

6.4.2 Contraintes sécuritaires

Les contraintes sont intermédiaires entre les réseaux de la domotique et ceux des appareils médicaux. Une entreprise étant plus fortement hiérarchisée qu'une cellule familiale, chaque entité du réseau Jini ne doit pas être certifiée comme nous pourrions le faire pour un membre de la famille. La notions de « groupe de services » et « d'authentification des utilisateurs » sont donc toutes les deux appropriées à ce type d'application. Remarquons que ce type de réseau a l'avantage d'être géré par une ou plusieurs personnes concernées (les administrateurs), que sa structure est relativement fixe. L'administration de droits d'accès et autres gestions sécuritaires s'en trouve dès lors facilitée.

6.5 Les réseaux ouverts offrant des services

Parmi les réseaux ouverts, Internet est celui que nous connaissons tous. Les caractéristiques principales de ce réseau ainsi que ses lacunes sécuritaires ont déjà été brièvement soulevées dans la section 3.1.

6.5.1 Quelques exemples

Une société décide d'offrir un ensemble de services Jini à ses clients. Ceux-ci sont enregistrés dans le lookup de la société mais également dans d'autres lookups appartenant à des tiers. La valeur du service est le plus souvent exécutée sur la machine de la société.

Prenons un autre type d'exemple : celui des systèmes d'interconnexion de chaînes d'hôtels et autres grandes corporations. Leur software tourne le plus souvent en DOS, ils ont une adresse IP changeante, et ils peuvent apparaître ou disparaître spontanément. Chacun de ces logiciels souhaiterait communiquer avec les autres mais leur hétérogénéité et la nature dynamique du réseau ainsi formé les handicapent fortement pour accomplir cette interopérabilité. Jini pourrait dès lors servir de couche de transport intermédiaire gérant la découverte et le bon maintien des communications entre clients et services (chaque client ou service étant une application « interface » pour chaque logiciel hétérogène).

6.5.2 Contraintes sécuritaires

Internet est par définition un réseau soumis à de nombreuses attaques potentielles. Il est évident que ce sont les extrémités communicantes qui doivent mettre en place les méthodes sécuritaires adéquates (à l'aide de la cryptographie, par exemple).

Dans le cadre de la diffusion de services payants par un opérateur commercial, il est capital de pouvoir également contrôler la visibilité de ces services et de contrôler les droits d'accès offerts aux clients.

Chapitre 7

Comparaison de travaux réalisés

Dans cette partie, nous analysons différents mémoires et travaux proposant une architecture pour rendre Jini sécurisé. Cette analyse s'effectue en deux phases : la première résume le travail entre autres par une rapide présentation de ses innovations et particularités, la deuxième vérifie l'adéquation du travail par rapport aux exigences sécuritaires que nous nous sommes définies dans la section 5.11.

Remarque : Tout au long de ce chapitre, nous réutilisons les notations définies dans la section 5.3. Nous en introduisons également de nouvelles au fil des articles exposés.

7.1 Article – « Trade-offs in a Secure Jini Service Architecture », DUT Munich, 2000

7.1.1 Explication du travail

Introduction et idées novatrices

Dans leur article [7], Peer Hasselmeyer, Roger Kehr et Marco Voβ ont imaginé une extension (cf. figure Figure 7.1) de l'architecture Jini permettant de résoudre les problèmes de visibilité et de contrôle d'accès aux services par les clients ainsi que le problème du proxy « opaque » (cf. section 5.10.4). Plus exactement, ils transposent ce dernier problème vers celui de l'obtention sécurisée d'un proxy de confiance. Pour réaliser cela, ils mettent en place un réseau de « Secure lookup » (SL) qui permet le transfert des proxies des services vers les clients par des moyens de contrôle d'accès et de communication sécurisée.

Le contrôle de l'accès et de la visibilité des services est accompli par les notions de « Secure Group » et de « Capabilities ». Les groupes sécurisés rassemblent des services nécessitant des droits d'accès identiques. Ils peuvent être hiérarchisés sous la forme d'arbre et par un principe de transitivité des droits, chaque autorisation pour un groupe donné dans la hiérarchie implique les mêmes autorisations pour tous ses sous-groupes. Les capabilities associent l'identifiant d'un acteur Jini à un ensemble de permissions d'accès aux services. Chaque permission correspond à un groupe cible et aux droits d'accès autorisés. Ces droits d'accès sont de deux types possibles : « *register access right* » et « *lookup access right* ». Autrement dit, il s'agit des droits d'enregistrement d'un proxy par un service ou de recherche de proxies par un client. Les capabilities sont fournies à la demande aux clients par un service Jini : le « Capability Manager » (CM). Celui-ci est enregistré dans un groupe spécial : le groupe « capability ». Ce groupe permet l'enregistrement des services CM exclusivement mais autorise le libre accès aux proxies de ces CM pour tous les autres clients du SL.

Pour procéder à l'authentification des différents acteurs, une troisième entité a été ajoutée : le « Certificate Authority » (CA). Celle-ci distribue des certificats numériques qui ne peuvent être délégués. Le CA est connu de tous, et donc en particulier sa clé publique afin d'authentifier les certificats des autres entités. Le CA distribue quatre types de certificats différents : un certificat pour le P_B de SL, un certificat pour le CM, un certificat pour les SL, un certificat pour les C et S.

Dans leur architecture (cf. Figure 7.1), le SL (avec son P), le CM et le CA forment ensemble les composants de base pour déterminer la confiance qu'un client doit avoir envers le proxy du service.

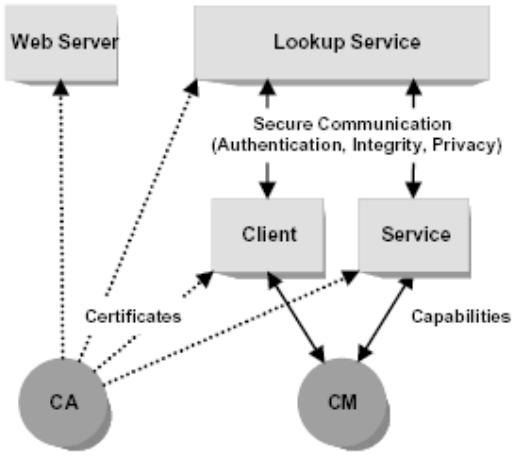

Figure 7.1 : Composants de l'architecture sécurisée

Pour garantir la confidentialité de la description des services, les communications entre le P de SL et le service SL lui-même sont chiffrées à l'aide de RMI over SSL. Il en va de même pour les communications avec le CM. Le protocole de découverte est légèrement modifié pour permettre le transfert sécurisé du proxy du SL au client C : lors de la réponse du SL à C, ce premier ajoute sa signature à la fin du proxy marshallisé. Le client est alors en mesure de déterminer qu'il s'agit bien du proxy d'un SL connu. Le protocole de découverte reste cependant compatible avec les anciennes spécifications.

Remarques :

(a) A l'exception des communications entre C et SL ainsi que C et CM, la sécurité des communications entre le proxy et son service est laissée au soin de l'implémentation du proxy !

(b) La notion de « Secure Group » est orthogonale à la notion de groupe définit par les spécifications de Jini. Elles ne sont pas identiques et peuvent être complémentaires. Pour garder la libre découverte des services, il existe un groupe sécurisé particulier du nom de « public » pour lequel les permissions sont court-circuitées.

(c) Attention, c'est la transmission du proxy du service au client qui est sécurisée dans cette architecture à travers le(s) SL(s) par des liaisons chiffrées.

Fiche caractéristique

But du travail :	Offrir une extension sécurisée de l'architecture Jini caractérisée par « un réseau de services lookups sécurisés », ceci afin de permettre : - L'acquisition fiable par le client des proxies de services (C est certain d'obtenir le P du service recherché). - Le contrôle de la visibilité et de l'accès aux services par les clients (au niveau du lookup).
Environnement cible :	Aucun spécifié : à priori, tout environnement mettant en œuvre des acteurs Jini. En pratique, la solution nécessite un réseau permettant une certaine centralisation : « réseau d'entreprise, réseau Internet, réseau d'un particulier,… »

Utilise :	Une implémentation personnalisée du service lookup (reggie) et des classes ayant attrait au processus de découverte, de nouvelles méthodes « `lookup` » et « `register` » pour l'interface « *`ServiceRegistrar`* », l'implémentation d'un service Jini CM, RMI over SSL.
Restrictions :	- Aucune sécurité prévue dans l'interaction classique du « client-proxy-service » en dehors de celle du « client-proxy-lookup » et du « client-proxy-CM ». - D'où l'hypothèse : « P implémente les contraintes sécuritaires nécessaires au domaine d'application de son service ». - Préserver une compatibilité totale avec les anciennes applications Jini comme : les S, C et L.
Caractéristiques :	- Un « Secure lookup » (SL). - L'adjonction d'un service Jini centralisé pour la gestion et la distribution des capabilities : le « Capability Manager » (CM). - L'adjonction d'une entité distributrice de certificats : le « Certificate Authority » (CA). - L'utilisation de « capabilities » : droits de recherche de services (*lookup access rights*) ou d'enregistrement de services (*register access rights*) que peut avoir un client d'un SL. - L'utilisation de quatre types de certificats différents : un certificat pour le P_B de SL, un certificat pour le CM, un certificat pour les SL, un certificat pour les C et S. - Notion de « groupes hiérarchisés de services » pour lesquels des clients peuvent avoir des « capabilities » différents. - Le protocole de découverte de services est légèrement modifié (mais reste compatible avec la version originale). - Administration centralisée (CM, CA et SL). - Le proxy est signé par le service (S ou SL) mais la vérification du certificat se fait par rapport à CA.

Résumé de l'architecture

Pour plus de détails sur le comportement des différentes entités, nous nous référons à l'article [7], section 6.

7.1.2 Adéquation par rapport à nos exigences

Ex. N°	*Adéq?*	*Détail*
(E1)	OUI	Le réseau sécurisé mis en place entre C et S par les SL nous assure que nous avons le bon P_S appartenant au bon groupe sécurisé. Mais cette constatation dépend essentiellement de la fiabilité des SL et de la bonne authentification effectuée entre les C et SL ainsi que les S et SL. Nous pourrions renforcer le mécanisme en transférant par la même occasion le certificat de S à travers les SL. Ceci nous permettrait également d'établir avec certitude la correspondance « S – F ». Quant à P_B, il est signé à l'aide du mécanisme de signature offert par Java. Il contient un certificat du service signé par le CA … qui est connu de tous.
(E2)	NON	Rien n'est prévu dans l'architecture mais P peut éventuellement implémenter le nécessaire.
(E3)	NON	Rien n'est forcément prévu dans l'architecture (voir difficultés E3).
(E4)	NON	Rien n'est prévu dans l'architecture.
(E5)	O/N	Sous l'hypothèse que tous les SL font bien leur travail et ne sont pas « piratés », S est certain que c'est un client autorisé qui utilise son P. Mais si l'hypothèse n'est pas vérifiée, alors n'importe quel mauvais client peut accéder à ses services (si P n'intègre pas un mécanisme supplémentaire d'authentification). De plus, l'hypothèse ne nous permet pas de savoir avec exactitude quel client a accès au service (excepté le fait qu'il soit autorisé).
(E6)	O/N	Reprenons l'hypothèse émise dans l'adéquation (E5) : dans ce cas, si C a confiance envers le P de S alors nous pouvons considérer qu'il parle à S. Mais il n'y a par-là aucune authentification (mutuelle) réalisée entre C et S au niveau de l'architecture.
(E7)	NON	Rien n'est prévu dans l'architecture.
(E8)	NON	Rien n'est prévu dans l'architecture.
(E9)	OUI	Sous l'hypothèse émise dans l'adéquation (E5), seuls les clients possédant les « capabilities » requises peuvent obtenir le proxy des services demandés. Ce sont les SL qui assurent ce contrôle d'accès.

(E10)	NON	Rien n'est prévu dans l'architecture. C'est au client de spécifier l'éventuelle « politique de sécurité » que doit utiliser sa JVM lorsqu'il reçoit le code venu de tel ou tel serveur.
(E11)	O/N	Aucun contrôle... excepté le fait que le client ait obtenu un proxy de confiance.
(E12)	OUI	Sous l'hypothèse de l'adéquation (E5), seuls les services possédant les « capabilities » requis peuvent enregistrer leur proxy dans les groupes spécifiés. Il suffit donc que le SL n'accepte de gérer que certains groupes parmi lesquels les services visés sont associés. Remarquons que cette association dépend du CM.
(E13)	NON	C peut contrôler l'accès que tout objet Java a sur ses ressources (via les permissions Java) mais c'est à lui qu'incombe la responsabilité de mettre en oeuvre tout le nécessaire. L'architecture ne propose rien, elle prend même pour hypothèse que P est « de confiance » ce qui signifie à la limite qu'il est autorisé de lui allouer tous les privilèges.
(E14)	NON	La recherche de service et la notion de groupe sécurisé permet de préciser les services qu'un client peut atteindre. Or, notre exigence fait référence à une atomicité au niveau des méthodes de proxy. L'architecture n'offrant aucun moyen d'identification de C auprès de S, celui-ci n'a pas non-plus la possibilité de modifier son comportement en fonction de l'identité de son client (voir aussi la remarque en bas du tableau).
(E15)	NON	Il n'y a rien dans l'architecture résolvant ce problème et encore moins dans la plate-forme Java 2 sous-jacente.
(E16)	NON	Il n'y a rien dans l'architecture résolvant ce problème et encore moins dans la plate-forme Java 2 sous-jacente.
(E17)	OUI	La compatibilité est totale. Les SL prennent en charge les anciens S et C dans un groupe sécurisé (« public ») aux permissions ouvertes à tout le monde. Le protocole de découverte, même s'il est modifié, ne rajoute que des données supplémentaires lors du transfert du proxy du lookup au client, ces données n'étant pas prises en compte par les anciens clients.
(E18)	OUI	Les notions en sécurité impliquées sont minimes.
(E19)	OUI	L'architecture est simple à comprendre et le programmeur n'est pas confronté à des mécanismes sécuritaires complexes. A quelques exceptions près, il développe des applications Jini comme il le faisait avant (il doit juste ajouter ses « capabilities » dans les méthodes « `lookup` » et « `register` » normalement standard aux proxies des services).

(E20)	OUI	Si nous développons des applications Jini pour un réseau plus ou moins fixe d'ordinateurs conventionnels, alors les performances ne devraient pas être défaillantes. Mais dans le cas de réseaux à caractère plus « spontané » et « mobile », la centralisation de certaines entités et la notion de groupe risque de gêner l'utilisation d'une telle solution.
(E21)	O/N	Puisque l'architecture ne propose ou n'impose aucune procédure sécuritaire pour les échanges d'informations « proxy – service », l'indépendance de protocole reste totale à ce niveau. Cependant, pour ce qui est des communications « P – SL » et « P – CM », le protocole imposé par l'implémentation nécessite que le client possède RMI over SSL !
(E22)	O/N	La « libre » découverte des services est automatiquement en conflit avec la notion de « groupe » et de « restriction d'accès au niveau du lookup ». Néanmoins, les SL sont implémentés pour rester compatibles avec les L non-sécurisés (grâce au groupe sécurisé « public »). La libre découverte est donc préservée pour les C et S ne désirant pas faire partie de groupes sécurisés.
(E23)	OUI	La notion de groupe ne semble pas être en conflit avec l'aspect « d'auto réparation » offert par la technologie Jini : nous pouvons toujours pratiquer la réplication de plusieurs SL, la gestion des réservations, …

Remarque concernant l'adéquation (E14) : Dans la mesure où les groupes de sécurité sont hiérarchisés et que les permissions d'accès à un groupe plus haut dans la hiérarchie impliquent les mêmes permissions pour ceux qui dérivent de ce groupe, nous pourrions imaginer un mécanisme capable d'offrir la granularité d'accès aux méthodes du proxy : il s'agirait de créer autant de sous-groupes, d'interfaces de proxy et d'implémentations de proxies qu'il y a de combinaisons entre les différentes méthodes du proxy enregistré. Mais cette solution me paraît peu réaliste en vue de l'espace mémoire requis pour entreposer ces proxies, du nombre de versions à gérer (qui risque vite d'exploser) et enfin de par le fait qu'une structure en forme de graphe serait plus appropriée que la structure en forme d'arbre ici proposée.

7.1.3 Appréciations personnelles

Les avantages

- La solution offre l'avantage de faciliter l'administration des droits d'accès aux services puisqu'elle est centralisée autour du CM.

- La gestion des droits d'accès (capabilities) est peut-être centralisée, mais le contrôle d'accès quant à lui est réparti entre les différents SL puisqu'ils ne desservent qu'un sous-ensemble de tous les clients potentiels et que ce sont les clients qui viennent avec leurs capabilities ! Les LS ne doivent donc pas posséder pour tous leurs clients une matrice d'accès leurs autorisant tel ou tel service. Ils doivent essentiellement posséder la clé publique du CM et celle du CA pour vérifier les capabilities.

- La solution limite fatalement l'aspect dynamique d'une fédération Jini de par les exigences administratives. Mais comme nous l'avons déjà fait remarquer pour l'adéquation (E22), le groupe sécurisé « public » permet de préserver la libre découverte des services pour les acteurs non-sécurisés.

- La solution est compatible avec les anciennes spécifications à l'exception près qu'un ancien client ne pourra pas accéder à un service appartenant à un groupe sécurisé.

- La solution permet le contrôle de la visibilité et de l'accès aux services. Ceci est une exigence de taille pour des réseaux où la simple présence d'un service chez un fournisseur concurrent peut déjà avoir des répercutions commerciales.

- La solution modifie le protocole de découverte de façon à vérifier l'authenticité du proxy du lookup avant même qu'il soit instancié !

- Les quatre différentes catégories de certificats permettent d'éviter qu'une entité ne simule une autre sans les autorisations appropriées.

Les inconvénients

- La solution est fortement dépendante de la présence d'entités fixes sur le réseau qui doivent être connues de tous ! Cette contrainte va à l'encontre de certains environnements d'applications (cf. section 7.2). Entre autres, elle nécessite une forte configuration initiale pour les différents acteurs du réseau.

- L'hypothèse sur laquelle ils basent la confiance du proxy est assez dangereuse. Il est clair que passer par un réseau « sécurisé » de lookups est une bonne idée pour obtenir un bon proxy. Mais lors du transit d'un proxy dans un SL intermédiaire, rien ne l'empêche d'être modifié par un BlackHat simplement parce que la machine hébergeant le SL en question n'est pas suffisamment protégée ! Je pense que le mécanisme doit être complété par le transfert obligatoire du certificat du service, de la signature du contenu du proxy par le service et également par des mécanismes sécuritaires pour les communications entre le proxy et le service.

- La solution prend indirectement pour hypothèse que les clients et les services exploitant l'infrastructure et possédant les capabilities appropriées, sont tous honnêtes par essence. Mais si par hasard, un seul de ces clients ne l'était pas, il pourrait obtenir les proxies de tous ses services autorisés et les distribuer « gratuitement » à d'autres clients.

- La sécurité de l'architecture est fortement dépendante de la bonne utilisation des certificats distribués par le CA (voir adéquation (E1)).

- L'architecture ne semble pas exploiter les mécanismes de domaines de protection et permissions d'accès offerts par la plate-forme Java 2. Le proxy instancié sur la machine du client est considéré comme fiable et se retrouve avec tous les droits que lui offre un client naïf.

- Aucun mécanisme concernant la révocation des certificats n'a été évoqué dans l'architecture. Mais cela ne devrait pas poser un problème majeur vu les hypothèses de base : le CA est connu de tous, il pourrait servir à la distribution de listes de révocation.

- La solution n'indique qu'un seul CA et qu'un seul CM. Dans le cadre d'un réseau plus important, comme Internet, et éventuellement hiérarchisé, il pourrait être favorable d'avoir plusieurs de ces entités qui coopéreraient entre elles. Notamment dans le cadre des réseaux ou certaines entités sont mobiles. Il serait intéressant qu'elles puissent dialoguer avec un CM plus proche qui serait responsable de contacter le CM original de l'utilisateur mobile. Cette idée de solution est évidemment à l'image des réseaux cellulaires de téléphones portables et de son standard actuel : le système GSM.

- De plus, le CA et le CM étant uniques sur tout le réseau, ils correspondent à un point d'attaque stratégique pour rendre l'architecture sécuritaire inopérante.

7.2 Mémoire - « Security in the Jini networking technology : a decentralized trust management approach », HUT, 2001

7.2.1 Explication du travail

Introduction et idées novatrices

Dans son mémoire [5], ***Pasi Eronen*** a imaginé et implémenté une architecture sécuritaire entièrement décentralisée : elle ne nécessite aucune entité centralisée sur le

réseau pour gérer la sécurité (voir remarque étape 4 ci-dessous). Il est dès lors normal que son environnement cible soit celui des réseaux ad hoc, en particulier celui des réseaux ad hoc de mobiles. Pour arriver à cela, l'auteur utilise les concepts de « gestion de confiance décentralisée » et de « délégation des permissions d'accès » : Chaque acteur (client, service, utilisateur,…) du réseau Jini est représenté exclusivement par une clé publique. La notion de nom ou tout autre attribut sécuritaire permettant d'identifier un de ces acteurs est reporté en dehors de l'architecture. Un service peut émettre un certificat associant un ensemble de permissions d'accès à une clé publique. Il parle dans ce cas de « certificat d'autorisation ». Les permissions d'accès du certificat peuvent être des permissions Java ou des droits d'utilisation du service en question. Par son acte de signature, le service délègue cet ensemble de permissions à la clé publique (K_P) d'un autre acteur (par exemple, un administrateur). Cet acteur est normalement détenteur de la clé privée (K_C) correspondante à K_P. Ayant hérité de ces privilèges, l'acteur est en mesure de déléguer lui aussi les permissions ainsi reçues ou du moins un sous-ensemble de celles-ci. Nous pouvons continuer le raisonnement de manière récursive. Au bout de la chaîne, nous avons l'utilisateur U qui possède donc normalement un ensemble de permissions envers les différents services du réseau. Pour prouver au service son droit d'utilisation, il devra lui fournir la « chaîne de certificats d'autorisation » qui le lie à ce service. Pour garder l'indépendance de protocole entre le proxy et le service, le transport de ces informations sera réalisé par le proxy lui-même. Le client C doit donc lui déléguer les permissions (un sous-ensemble des permissions de U) nécessaires pour accéder au service. De la même manière, un utilisateur U peut déléguer un sous-ensemble de ses droits aux applications de son choix sur base du niveau de confiance qu'il leur porte.

Remarque : L'auteur utilise un format particulier de certificats du nom de SPKI (Simple Public Key Infrastructure). Ceux-ci ont les caractéristiques suivantes : le sujet est identifié exclusivement par sa clé publique, le certificat incorpore des droits d'accès, il permet la délégation de ces droits et de former ainsi une chaîne de certificats. Pour plus d'informations, je vous invite à consulter le mémoire [5], section 2.2.2.

Fiche caractéristique générale

But du travail :	Imaginer et implémenter une architecture sécuritaire pour Jini qui soit complètement décentralisée.
Environnement cible :	Les réseaux ad hoc (en particulier d'appareils mobiles).

Utilise :	La sécurité de la plate-forme Java 2, ses extensions (JSSE), des librairies pour la manipulation de certificats SPKI, une extension RMI over TLS.
Restrictions :	- Il se focalise sur la relation « client – proxy – service ». - Il ne modifie pas l'implémentation de Jini telle qu'elle est fournie par Sun Microsystems.
Caractéristiques :	- Décentralisation complète : absence de toute entité sécuritaire centralisée (comme un CA, par exemple). - Certificats d'autorisation (≠ certificats de nom). - Gestion de confiance décentralisée (par chaîne de certificats). - Délégation des permissions Java ou autres entre C, P et S. - Chaque acteur Jini est exclusivement représenté par une K_P. - Distingue clairement les permissions d'accès Java 2 de C, P et S (domaines de protection différents). - Préserve l'indépendance de protocole (entre P et S). - Aucune sécurité au niveau du lookup (qui est inchangé). - Administration décentralisée. - Le proxy est signé par le service qui est lui-même l'émetteur du certificat.

Résumé de l'architecture

Pour décrire l'architecture (voir Figure 7.2), nous gardons les notations utilisées jusqu'à maintenant en particulier celles détaillant les acteurs des interactions Jini (section 5.3). L'architecture est ici expliquée simplement de façon à en percevoir son fonctionnement essentiel. Pour plus d'informations sur l'implémentation et autres difficultés rencontrées, je vous invite à consulter l'article [6] ou le mémoire [5].

L'architecture est caractérisée par plusieurs composants :

- **CSM** : « Client Security Manager »

  Il est responsable du contrôle d'accès aux clés privées de l'utilisateur, de l'authentification de P et de faire respecter les contrôles d'accès au niveau des

applications. Il offre au proxy téléchargé la possibilité de lui signer des données et de lui déléguer des permissions de l'utilisateur.

- **SSM** : « Server Security Manager »

  Il est responsable de vérifier la chaîne de certificats SPKI reçue de P. Il offre au service la possibilité de signer son proxy.

- **CR** : « Certificate Repository »

  Il fournit un moyen simple pour entreposer les certificats d'autorisation SPKI. Il peut être étendu pour supporter l'extraction des certificats depuis le réseau (via DNS, LDAP,…).

Voici d'autres notations :

- $\mathbf{K_{PT}, K_{CT}}$: La clé publique et la clé privée temporaires prêtées à P (par C).

- $\mathbf{K_{PS}}$: La clé publique du service S.

- $\mathbf{K_{PS'}}$: La clé publique du service S comme le prétend P.

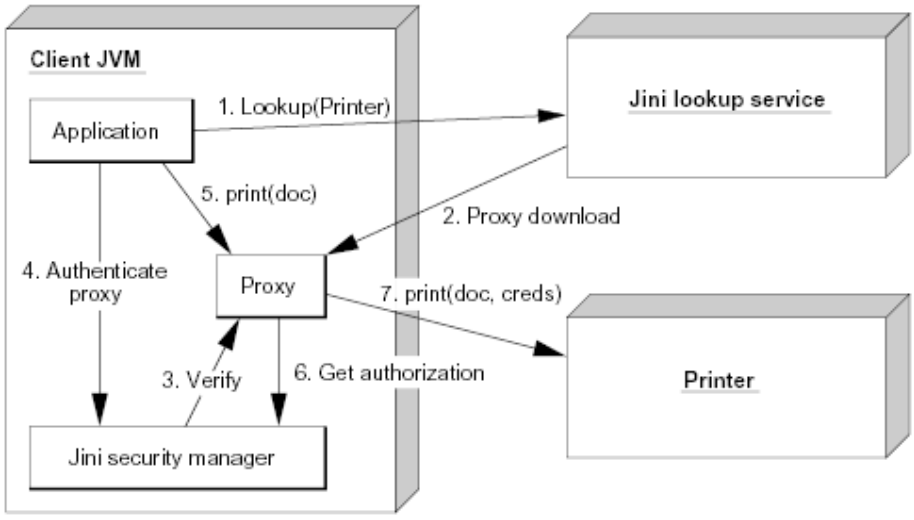

Figure 7.2 : Accès à un service Jini selon la nouvelle architecture sécurisée

Pour décrire l'architecture, imaginons un service « Printer » (S) offrant la possibilité à un certain client (C) d'imprimer un document. La transaction s'effectue en sept étapes.

Etape 0 :

Initialement, S doit signer le Bytecode (P_B) et l'état (P_S) de son proxy (P). Pour des détails d'implémentation, P_B est signé à l'aide d'une clé privée générée temporairement. Celle-ci est alors détruite. La signature est insérée dans le fichier JAR correspondant à P_B : ceci est une fonctionnalité standard offerte par Java. Le service crée également un certificat SPKI validant la clé publique générée à l'aide de sa clé privée (K_{PC}). Ce certificat est inséré parmi les champs de données de P_S. La signature de P_S est quant à elle, un peu différente : S demande au proxy de lui fournir son propre résumé de message. S peut alors ajouter, dans les champs de données de P_S, un certificat SPKI liant la clé publique du service (K_{PS}) au résumé de message fournit par le proxy.

Etape 1 à 2 :

Les deux premières étapes sont conformes au fonctionnement habituel de Jini : Notre client (C) obtient du service lookup (L) le proxy sérialisé (P_S) du service (S). Par la même occasion, il télécharge le bytecode du proxy (P_B) depuis le serveur de fichiers indiqué dans l'URL de P_S. C pense donc avoir obtenu le proxy (P) du service résultant de l'adjonction de P_S et de P_B.

Etape 3 :

- Le CSM demande à P la K_{PS}.
- P renvoie $K_{PS'}$ au CSM.
- Le CSM vérifie la signature de P_S et de P_B grâce à $K_{PS'}$.

  $\Rightarrow$ Il prouve ainsi que P a été signé par la $K_{PC'}$ correspondante à la $K_{PS'}$ reçue de P.

  $\Rightarrow$ Il ne prouve pas que $K_{PS'} = K_{PS}$!

Etape 4 : (optionnelle)

Pour prouver l'égalité « $K_{PS'} = K_{PS}$ », il faut soit que nous possédions déjà la K_{PS} originale, soit que nous l'obtenions par un moyen externe. Ceci induit la réception d'un certificat liant la K_{PS} à l'identité que nous utilisons pour représenter le service. Ce certificat peut être obtenu par le lookup, un CA ou par P lui-même.

Remarque : L'architecture n'est donc pas totalement décentralisée.

Etape 5 :

C tente d'utiliser une méthode de P (comportement standard).

Etape 6 :

- P demande au CSM de lui déléguer les autorisations nécessaires pour accéder à S.
- Le CSM vérifie que P tente réellement d'accéder à S.
- Le CSM vérifie que C a le droit de déléguer des permissions au nom de U.
- Le CSM génère une K_{PT} et une K_{CT} et délègue à K_{PT} les autorisations nécessaires. Cette clé publique temporaire est créée pour distinguer P de C et lui attribuer des permissions Java différentes de celles du client. Les domaines de protection Java entre C, P et S sont donc différents.
- Le CSM donne la possibilité à P de chiffrer des informations avec ces clés temporaires mais il ne lui fournit pas la clé secrète ! Il lui fournit un « manche » pour signer des données.
- Le CR cherche les certificats nécessaires pour former la « chaîne de certificats » liant C à S et les donne à P.

Etape 7 :

P doit établir une connexion sécurisée avec S à l'aide du « manche » que lui a fournit le CSM.. Il commence par s'authentifier auprès du service ou plus exactement, il lui prouve qu'il « possède » la K_{CT} correspondante à la K_{PT} qui le représente (protocole d'authentification classique). Une fois l'authentification effectuée, le canal de communication peut être chiffré et la liaison est donc sécurisée. En fait pour réaliser tout cela, P implémente le protocole de son choix (par exemple RMI over TLS). Il est ensuite chargé de prouver ses droits d'accès au service en lui transmettant la chaîne de certificats qu'il a reçue de C.

S passe alors la K_{PT} de P et la chaîne de certificat à son SSM qui vérifie l'exactitude de ces informations.

7.2.2 Adéquation par rapport à nos exigences

Ex. N°	*Adéq?*	*Détail*
(E1)	OUI	A l'étape 3, le CSM vérifie l'intégrité de P et à l'étape 4, il nous permet de connaître la K_{PS} du service.

(E2)	OUI	A l'étape 7, ils procèdent à une phase d'authentification qui peut ou non être mutuelle.
(E3)	NON	S sait qu'il parle à une entité représentée par une K_{PT} et il sait que cette entité a reçu des droits d'accès de C. Mais il n'a pas la possibilité de savoir que c'est son véritable proxy.
(E4)	NON	Si P n'a pas été modifié en cours de route, alors le mécanisme impliqué pourrait lui permettre de vérifier les certificats d'autorisation présentés par C. Mais cela n'est évidemment pas prévu dans l'architecture et c'est difficilement réalisable.
(E5)	OUI	A l'étape 7, P fournit les certificats d'autorisation à S. Ceux-ci lui permettent de connaître la clé publique de C. La clé temporaire K_{PT} fait partie de la chaîne de certificats et permet l'authentification de P auprès du service (via le « manche »). Si le service connaissait initialement la clé publique de l'entité qu'il considère comme C, alors il est en mesure d'authentifier l'utilisateur C de son proxy.
(E6)	O/N	A l'étape 6, le CSM vérifie que P tente réellement d'accéder à S. En fait, cette vérification consiste à ajouter dans le certificat SPKI délégué au proxy, le résumé de message de la K_{PS} du service destinataire. Ainsi, S pourra vérifier que c'est bien lui le destinataire du certificat et pas un autre service. Ceci semble plutôt être un mécanisme pour d'avantage protéger le service que le client. Dans cette exigence (E6), C cherche à authentifier S. Or, à regarder de plus près, c'est P qui se charge de cette authentification. La réponse à la question « C authentifie-t-il S ? », nous répondons « Oui » si C a une totale confiance en P ou nous répondons « Non » si ce n'est pas le cas.
(E7)	O/N	P a tout ce qu'il lui faut pour remplir cette exigence. Mais, tout dépend de la notion de confiance que porte C envers P.
(E8)	OUI	Si S a confiance en son client C et qu'il a créé un P capable de d'utiliser le « manche » du CSM alors ses données, lors de leur transfert, resteront confidentielles.

(E9)	NON	Le seul organe capable de contrôler la distribution des proxies est le lookup Service. Or celui-ci est resté inchangé par rapport aux spécifications Jini. Une autre solution serait de chiffrer le proxy de façon à ce que seuls les clients concernés puissent le déchiffrer. Mais cela risque de poser des problèmes par rapport à Java et au fonctionnement de Jini (un proxy implémente normalement une interface et non pas l'objet Java chiffré). De plus, avec la cryptographie à clé asymétrique, par exemple, la clé publique qui serait nécessaire pour restaurer le proxy devrait être gardée par un ensemble de clients concernés. Garder le secret deviendrait alors plus difficile. Ou bien dans le cas de l'utilisation d'une seule clé par client, il faudrait autant de proxies chiffrés qu'il y a de clients.
(E10)	OUI	P est lancé dans un domaine de protection Java distinct de C. Les permissions d'accès Java 2 qui lui sont attribuées sont donc différentes de C. En particulier celles qui touchent à l'accès aux ressources comme le système de fichiers. Le CSM joue également le rôle d'intermédiaire pour protéger l'accès aux clés privées de C.
(E11)	NON	Le client n'a aucun contrôle sur ce que fait P des données confidentielles que C lui aurait donné. Il doit lui faire confiance.
(E12)	NON	L est resté inchangé.
(E13)	OUI	Cf. (E10).
(E14)	OUI	S peut tout naturellement modifier son propre comportement en fonction du certificat d'autorisation que lui propose C (par l'intermédiaire de P). L'architecture permet donc de modifier cela mais au niveau de l'application du service. Le contrôle d'accès aux méthodes de P peut également être réalisé (mais exclusivement au niveau de S) car les permissions du certificat d'autorisation peuvent représenter des appels de méthodes de proxy.
(E15)	NON	Il n'y a rien dans l'architecture résolvant ce problème et encore moins dans la plate-forme Java 2 sous-jacente.
(E16)	NON	Il n'y a rien dans l'architecture résolvant ce problème et encore moins dans la plate-forme Java 2 sous-jacente.

(E17)	O/N	Un S et son P sécurisés doivent clairement être implémentés de façon à correspondre à l'architecture spécifiée. Un ancien client Jini ne pourra pas utiliser un service sécurisé par cette architecture. Un ancien service Jini pourra quant à lui théoriquement être utilisé sur un client sécurisé mais le service qu'il représente ne sera évidement pas sécurisé (donc le client non plus). Le lookup reste totalement compatible puisqu'il n'a pas été modifié. Ses protocoles restent de vigueur.
(E18)	NON	Le développement est facilité notamment par l'intermédiaire de modules de vérification de certificats, etc… Mais, surtout du côté de P et de S, nous ne pouvons pas dire que le développement d'applications Jini sécurisées se fait sans connaissances suffisantes en sécurité.
(E19)	OUI	L'architecture fournit des méthodes qui soulagent le développement des applications sécurisées.
(E20)	O/N	Dans le cadre des réseaux ad hoc de mobiles, nous pouvons premièrement déplacer la question de cette exigence : « Jini est-il une technologie que nous pouvons intégrer dans les appareils embarqués ? » Cette question nécessiterait une thèse à elle-seule. Les mesures effectuées par Pasi Eronen pour justifier les performances de son architecture sont raisonnables (d'autant plus que l'implémentation n'est pas optimisée) mais celles-ci ont été réalisées sur un PC conventionnel, ce qui ne représente pas vraiment un appareil embarqué et qui de plus est mobile. La réponse à notre exigence, est selon moi, positive mais pour un futur proche, quand les performances de ces appareils et des moyens de transmission sans fil seront plus appropriées aux besoins de Jini.
(E21)	OUI	Nous constatons qu'à l'étape 7 même l'authentification et le chiffrement du canal sont laissés aux soins du proxy.
(E22)	OUI	Le lookup est préservé dans sa spécification originale.
(E23)	OUI	Le principe du « Lease » n'a pas été retiré et ne semble pas être incompatible avec l'architecture présentée.

7.2.3 Appréciations personnelles

Pasi Eronen met en avant une approche totalement différente de celles généralement constatées dans les systèmes informatiques classiques. Ceux-ci basent généralement leur contrôle d'accès sur l'authentification de leurs utilisateurs. Ces systèmes

possèdent un moyen de connaître la totalité des droits d'accès qu'ils attribuent pour tous leurs utilisateurs. Or, ici, l'auteur décentralise complètement cette gestion de droits d'accès de façon à ce qu'ils ne soient plus entreposés et gérés par un seul organe mais par les utilisateurs eux-mêmes ou du moins certains d'entre eux. L'approche est très intéressante surtout dans le cas de systèmes distribués comme les réseaux ad hoc qui sont par définition des réseaux où les entités ne connaissent rien au préalable des autres entités composant le réseau.

L'administration des certificats dans sa solution, semble évidemment ardue. Et si nous souhaitons étendre son mécanisme pour des réseaux plus généraux (comme Internet), il s'agit de mettre en avant un ensemble de logiciels permettant de contrôler la diffusion des certificats. Dans le même ordre d'idées, la relation de confiance qui résulte de la délégation de permissions peut être la source d'une faille sécuritaire potentielle. Il suffirait d'un maillon faible et toute la relation de confiance s'écroule.

D'un point de vue technique, le proxy reçu du lookup est instancié dans la JVM du client avant qu'il soit totalement authentifié comme venant du service désiré. Ceci présente une faille de sécurité dans le sens où la méthode « constructeur » du proxy peut déjà posséder du code malicieux. Evidemment, si le proxy est instancié avec un domaine de protection extrême comme celui du « Sandbox », alors nous pouvons supposer qu'il ne pourra pas faire grand chose de nocif envers le système du client.

L'idée d'utiliser exclusivement une clé publique pour représenter chaque acteur Jini au sein de son architecture, me semble excellente car lorsque nous nous plaçons au niveau des communications réseau, une clé suffit amplement pour identifier toute entité logicielle ou matérielle. Le problème de la correspondance entre l'identité de ce niveau et celle que les êtres humains se font, est reportée à un niveau supérieur : celui de l'application, permettant en autres l'utilisation d'interfaces graphiques pour faciliter le dialogue homme-machine. De plus, l'auteur ajoute des modules facilitant la tâche du programmeur dans ce sens.

Pasi Eronen arrive à préserver l'indépendance de protocole (presque totale) entre le proxy et son service. Je dis « presque » car l'architecture impose certains comportements au niveau du proxy comme : utiliser l'API sécuritaire du CSM (du client), procéder à l'échange des certificats, … Mais n'est-ce pas là le prix de la sécurité ?

L'idée de donner la possibilité au proxy d'utiliser des clés cryptographiques à travers un API standardisé me paraît également très ingénieuse. Ceci donne tous les outils nécessaires au proxy pour s'authentifier et créer un canal chiffré entre lui et le service sans qu'il n'ait à un seul moment accès à une quelconque donnée confidentielle de C.

La question de savoir quels droits donner à un proxy arrivé même s'il vient d'un service que nous connaissons reste mystique. Devons-nous le considérer comme totalement fiable ou devons-nous lui permettre de n'accéder qu'à certaines ressources ? Cependant une solution est présente dans l'architecture car le proxy demande à C de lui fournir les permissions dont il a besoin. Ceci peut évidemment faire intervenir l'utilisateur qui peut décider du bon sens de cette demande. En fait, dans son architecture, le mécanisme de « permissions » présent dans la sécurité de la plate-forme Java 2 s'étend naturellement aux systèmes distants (un service devenant une ressource, pour le client, comme n'importe quelle autre de ses ressources locales).

Nous avons donc encore un autre point positif pour cette architecture. Mais il serait sûrement très intéressant d'automatiser le processus de recherche des bonnes permissions pour un certain proxy de service via, par exemple, des « permissions types » en fonction du domaine ou de la catégorie du service. D'autres idées comme la certification du style « PGP » serait également une voie d'exploration pour ce genre de déploiement : un ensemble d'utilisateurs en qui nous aurions confiance affirment que tel service n'a besoin que de telles permissions.

Le mécanisme de délégation est très bien venu également dans les réseaux de services. Imaginons qu'un service doive utiliser un autre service pour exécuter une sous-tâche de son travail mais il n'a malheureusement pas les droits pour l'utiliser. Le principe de délégation nous permet, en tant que client, de lui déléguer le droit d'accéder à ce service en notre nom et ce, pendant une session, une période ou un nombre de fois données. Le certificat d'autorisation transmis peut inclure une information interdisant le service de déléguer à son tour les pouvoirs ainsi transmis.

La gestion de la révocation de certificats n'est pas claire dans l'architecture qu'il nous propose. Où et comment les différents services peuvent prendre connaissance des certificats d'autorisation que des clients auraient fournis à un proxy et qui pour une raison comme celle d'un proxy malicieux, par exemple, devraient être révoqués ?

Placer la sécurité au niveau de l'interaction « client-proxy-service » offre un avantage redoutable : nous pouvons étendre ce mécanisme à n'importe quel service, le lookup Service compris. Nous pouvons même authentifier le lookup à l'aide d'une entité tierce de confiance et s'assurer qu'il est bien le lookup « sécurisé » en qui nous avons confiance. Tout mécanisme de sécurité visant par exemple à limiter l'accès aux proxies de certains services, peut être implémenté par-dessus cette architecture. Cependant, il reste le problème de l'activation du proxy du lookup dans la JVM du client avant qu'il soit authentifié. Le processus de découverte peut être modifié de la même manière que celle présentée dans l'article de la section 7.1.

7.3 Le projet « Davis », 2001

La communauté des développeurs Jini possèdent un site WEB [14] leur offrant la possibilité d'initier et de superviser des projets de développement pour cette technologie. C'est dans ce contexte qu'est né le projet « Davis » lancé par Bob Scheifler, architecte renommé de chez Sun Microsystems, et par Timothy Blackman. Ce projet est intégré au sein de la « Jini technology project team ». Il tente de répondre au besoin d'une architecture sécuritaire pour la technologie Jini. Le projet Davis a produit à l'heure actuelle « l'Overture 0.05 » [17] qui est la deuxième version publique de l'implémentation, après « l'Overture 0.04 ». Ces versions sont destinées à valider l'architecture par la communauté des développeurs Jini. Le projet Davis aboutira dans une prochaine version du « Jini Starter Kit » distribuée par Sun Microsystems.

Le projet Davis étant une API qui a fortement évoluée au cours de cette année et n'étant pas basée sur un travail scientifique rendu public, une étude de son architecture nécessiterait un mémoire à lui seul. Néanmoins, nous explicitons ici, dans un but de complétude, quelques solutions principales ou innovantes développées par les ingénieurs de chez Sun Microsystems.

Jini ses besoins sécuritaires, vus par le projet Davis

Le projet Davis fournit une liste exhaustive de ses objectifs d'implémentation [16].

La plate-forme Java est fournie avec un ensemble de mécanismes sécuritaires intégrés visant à protéger la machine virtuelle (JVM) du code téléchargé. Mais, Java, RMI ou Jini n'ont aucun concept standardisé en ce qui concerne les communications réseau. Le premier but du projet Davis est donc d'apporter à Jini les notions standard de « sécurité réseau » (authentification, confidentialité, intégrité, …).

Le projet Davis tente également d'apporter une solution au problème du code mobile déjà évoqué dans la section 5.10.4. Voici pour rappel les gros titres :

- Authentification et autorisation mutuelle (client – service).
- Etablissement de la notion de confiance (client – proxy).
- Préservation de l'intégrité totale des objets transmis (dont le proxy).

Solutions émises par le projet Davis

Pour résoudre le problème de la confiance à attribuer au proxy, le projet Davis distingue, comme le fait déjà le modèle sécuritaire Java, le code local du code distant (téléchargé à la demande). Le code local qui est celui faisant partie de notre système

de fichiers est considéré comme fiable au sens sécuritaire. A l'opposé, le code distant est considéré comme potentiellement dangereux.

La première idée est donc de se dire : « si un objet téléchargé, n'utilise que du code et des objets locaux et que ses attributs semblent corrects, alors cet objet est « de confiance ». Pour s'assurer de cela, il faut transformer l'objet reçu en un graphe informatique et utiliser un algorithme de parcours en profondeur pour déterminer que toutes les sous-classes utilisées éventuellement dans l'objet sont locales.

Ceci ne résout pas le téléchargement de code dynamique, en effet. Si la confiance est transitive, alors nous pouvons utiliser le raisonnement suivant : « Si je fais confiance au fournisseur du service alors je fais confiance à son service et si je fais confiance à son service alors je fais confiance à son proxy ». Le véritable problème à résoudre sous cette hypothèse est de déterminer si le proxy est bien celui du service que nous souhaitons utiliser. Pour cela, la deuxième idée est d'utiliser un mécanisme de « bootstrapping ». Le proxy, pour lequel nous devons effectuer la vérification, comprend un autre proxy appelé « bootstrap proxy ». Celui-ci n'implémente qu'une seule méthode permettant de joindre le service et n'utilise que du code local ! En utilisant la première idée développée plus haut, nous pouvons établir la confiance ou non du bootstrap proxy. L'unique méthode implémentée, permet de recevoir du service, un objet (« Verifier Object ») qui sera utilisé pour valider l'intégrité du proxy (données et code). Il est clairement nécessaire que la liaison entre le client et le service soit sécurisée et que le service soit bien identifié par le client. Il est possible d'utiliser pour ce faire une connexion SSL garantissant ainsi que l'objet sérialisé « Verifier » est bien authentique. Mais il reste son code qui est téléchargé depuis un serveur de fichiers. C'est pour cela que le projet Davis introduit une troisième idée.

La troisième idée consiste à garantir l'intégrité totale des objets échangés. Le protocole de transfert de fichiers « class » le plus utilisé par les développeurs Jini est le protocole « HTTP ». En particulier, le codebase est une URL de type HTTP. Mais ce protocole n'offre aucune garantie d'intégrité des données. Deux solutions sont proposées. La première utilise une variante sécurisée : « le HTTPS ». Celui-ci garantit l'authentification du serveur, la confidentialité et l'intégrité des données transmises. Mais il y a certains inconvénients : il faut générer des certificats numériques pour l'authentification des parties ; il n'est pas toujours nécessaire de garantir la confidentialité du code transféré ; chiffrer et déchiffrer le code peut être lourd pour des clients plus légers en ressource CPU. L'autre solution consiste à utiliser un autre type de protocole et d'URL compatible avec l'ancien HTTP : il s'agit du « HTTPMD ». Celui-ci consiste à rajouter à la fin de l'URL un résumé de message (Message Digest) du fichier « jar » spécifié. Ce principe ne fonctionne que pour les fichiers « jar », un résumé de message étant difficilement calculable pour un

répertoire. L'algorithme utilisé pour calculer le résumé de message fait également partie de l'URL. Le codebase prend donc par exemple la forme suivante : « httpmd://monserveur:8080/proxy-dljar ; md5=xxx…x » où les croix sont à remplacer par le résumé de message à proprement parler. Remarquons que nous ne garantissons que l'intégrité du code dans ce cas-à. Nous pouvons cependant utiliser des serveurs HTTP normaux ! Ceci présente donc de nombreux avantages d'ordre pratique.

Du point de vue de la flexibilité, le projet donne la possibilité de paramétrer et d'inter-changer l'algorithme de « confiance du proxy » avec de nouvelles versions.

Quelques critiques personnelles (sans détails)

Les questions restées ouvertes (-)

- Pas de solution pour la gestion des groupes de services.
- Pas de solution pour le transfert des clés.
- Le lookup ne doit-il pas avoir des caractéristiques particulières ?
- Et donc par extension le protocole de découverte ?
- L'établissement de la confiance ne se base que sur la notion de localité du code (code local / code téléchargé).
- Il faudrait des outils plus poussés et flexibles pour l'administration.

Les bonnes idées (+)

- L'algorithme de vérification est une bonne idée surtout s'il est fortement paramétrable.
- Le projet se focalise sur le premier problème sécuritaire de Jini : la confiance envers le proxy.
- Les deux solutions : « HTTPMD » et « HTTPS ».
- Le mécanisme de « bootstrap proxy ».

Chapitre 8

Conclusion

Au cours de nos réflexions sur la sécurité nécessaire à l'infrastructure Jini, nous avons clairement établi ses trois caractéristiques principales ayant un impact direct sur la sécurité : il s'agissait du « code mobile », de « l'indépendance de protocole » et de « la libre découverte des services ».

L'analyse des deux solutions architecturales émises dans le chapitre 7 nous montre qu'il y a moyen de rendre Jini sécurisé. En particulier, pour ce qui est du transfert d'informations, les principes et méthodes sécuritaires propres au monde des réseaux (comme le chiffrement des données) sont bien entendu applicables aux réseaux d'acteurs Jini.

Cependant, Jini soulève de nouveaux problèmes sécuritaires liés à son mode de fonctionnement. Ceux-ci touchent essentiellement le client. Nous faisons référence au problème du « proxy opaque » induit par le « code mobile » et « l'indépendance de protocole ». Il est maintenant certain que toutes les solutions proposées se basent essentiellement sur la notion de confiance pour résoudre cette problématique. Le raisonnement est louable : un client désire utiliser les services offerts par un opérateur quelconque en qui il a confiance. A partir du moment où les différents acteurs (client, proxy et service) arrivent à s'authentifier, le client laisse le proxy du service faire son travail. Mais la confiance est toujours une notion relative. Plutôt que de donner les pleins pouvoirs au proxy, pourquoi ne pourrions-nous pas nuancer la notion de confiance ? La solution de Pasi Eronen (section 7.2) est probablement celle des deux qui se rapproche un peu plus de cette idée. Malheureusement, la notion de confiance est la seule solution viable à ce problème si nous voulons préserver les deux caractéristiques du « proxy opaque ».

D'autres travaux existent également et proposent des solutions quelques peu différentes. Nous faisons par exemple référence au mémoire [3], pour lequel Thomas Schoch valorise une architecture permettant l'authentification de l'utilisateur du client auprès du service ; ceci tout en préservant une transparence sécuritaire complète pour les applications clientes déjà développées. Mais nos deux travaux n'ont pas été choisis par hasard. En regardant de plus près chacune de ces architectures, nous constatons que, en dehors du problème du « proxy opaque », ces solutions ne visent qu'un sous-ensemble des fonctionnalités en sécurité nécessaires à toute catégorie d'application Jini. De plus, ces sous-ensembles sont relativement distincts puisqu'ils touchent des environnements d'application différents.

Jini est normalement par nature assez large en ce qui concerne ses champs d'utilisation. Les travaux futurs devront en conséquence tester les différentes architectures en fonction d'environnements d'application concrets et distincts. Une liste exhaustive des fonctionnalités sécuritaires obligatoires à chacun des ces domaines d'application devrait permettre de valider ou d'améliorer telle ou telle solution.

Mais au bout du compte, l'idéal serait d'obtenir une solution hybride, facilement paramétrable en fonction du contexte de déploiement. Le projet « Davis » s'oriente d'une certaine manière dans cette direction. Son analyse serait dès lors très intéressante à effectuer. Mais il est trop tôt pour se prononcer sur le sujet, le projet étant en perpétuelle modification et n'ayant pas encore abouti dans un « Jini Starter Kit ».

Pour finir, n'oublions pas que Jini incorpore également d'autres notions comme : les « Leases », les « Distributed Events » et les « Transaction ». Celles-ci nécessitent sûrement des solutions sécuritaires toutes particulières qu'il serait bon de mettre en valeur.

Bibliographie

[1] Dieter Gollmann. *Computer Security*. John Wiley & Sons, Inc., 1999.

[2] W. Keith Edwards. *Core Jini, Second Edition*. Prentice Hall, 2001.

[3] Thomas Schoch. *An Authentication and Authorization Architecture for Jini Services*. Diploma thesis, Darmstadt University of Technical. 2000. http://www.inf.ethz.ch/~schoch/da.pdf

[4] Thomas Schoch, Oliver Krone, Hannes Federrath. *Making Jini Secure*. 2001. http://www.inf.ethz.ch/~schoch/making-jini-secure.ps

[5] Pasi Eronen. *Security in the Jini Networking Technology : A Decentralized Trust Management Approach*. Diploma thesis, Helsinki University of Technology. 2001. http://www.cs.hut.fi/~peronen/publications/masters_thesis.pdf

[6] Pasi Eronen and Pekka Nikander. *Decentralized Jini security*. In Proceedings. 2001. http://www.cs.hut.fi/~peronen/publications/ndss_2001.pdf

[7] Peer Hasselmeyer, Roger Kehr, and Marco Voβ. *Trade-offs in a secure Jini service architecture*. 2000. http://www.ito.tu-darmstadt.de/publs/papers/usm00.pdf

[8] Jon Siegel. *Corba fundamentals and programming*. John Wiley & Sons Inc.,1996.

[9] Laura Lemay & Rogers Cadenhead. *Java 2 plate-forme*. Sam Publishing, 1999.

[10] H.X.Mell & Doris Baker. *La cryptographie décryptée*. Campuspress , 2001.

[11] Li Yong. *Inside Java 2 Platform Security : Architecture, API design, and implementation*. Addison-Wesley, 1999.

[12] Bruce Schneier. *Cryptographie appliquée*. International Thomson Publishing France, 1996.

[13] George Coulouris, Jean Dollimore & Tim Kindberg. *Distributed Systems concepts and design, Third Edition*. Addison-Wesley, 2001

[14] The community Resource for Jini Technology.
http://www.jini.org

[15] Sun Microsystems. *Jini Network Technology*.
http://www.sun.com/jini

[16] Bob Scheifler, Timothy Blackman. *Davis Project.* Jini community project.
http://davis.jini.org

[17] Bob Scheifler, Timothy Blackman. *Davis Project -Overture 0.05 Release*. Jini community project.
http://davis.jini.org/overture.html

[18] Object Management Group. *CORBA Security Service Specification*. Version 1.8. (document 02-03-11). 2002.
http://www.omg.org/technology/documents/formal/security_service.htm

[19] ITU-T Recommendation X.800 (1991). *Security architecture for Open Systems Interconnection for CCITT applications*. 1991.

[20] ITU-T Recommendation X.810 (1995). *Information technology - Open systems Interconnection - Security frameworks for open systems : Overview*. 1995.