MASTER EN SCIENCES INFORMATIQUES

Aspects de la sécurité dans l'environnement Java

Marchal, Joël; Renotte, Luc

Award date:
2002

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2001-2002

Aspects de la Sécurité dans l'environnement Java

Joël MARCHAL
Luc RENOTTE

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

RÉSUMÉ

Avec l'essor d'Internet, les applications distribuées se sont généralisées. Avec l'interconnexion des réseaux, les problèmes de sécurité sont devenus une préoccupation essentielle. Dans ce contexte, Java occupe une place prépondérante, avec la particularité d'intégrer le souci de sécurité dans la conception même du langage. D'un autre côté, force est de constater que la sécurité ne fait pas encore partie intégrante du processus de développement des logiciels.

Nous nous proposons d'investiguer les besoins en sécurité de manière générale, et les techniques particulières offertes par l'architecture de sécurité Java. Nous mettons en évidence les solutions originales du langage, telles que le chargeur de classe personnalisé, le contrôleur d'accès, les objets « signés, scellés, gardés », SSL.

Nous proposons ensuite de décrire une méthodologie basée sur UML, tenant compte de recherches en cours sur le sujet, telle que UMLsec.

Sur base de cette connaissance, nous proposons un guide de sécurisation d'un projet Java, qui montrera comment intégrer la sécurité dans un projet dès les premières phases de la conception. Nous illustrons notre approche par la mise en œuvre du guide dans une application exemple.

Soulignons qu'une démarche essentielle de ce mémoire a été de réduire la complexité du sujet dans une perspective pédagogique.

Mots-clés : sécurité, Java, systèmes distribués, modélisation, UML , UMLsec, développement

ABSTRACT

The expansion of the Internet has been accompanied by the widespread development of distributed applications. With network interconnectivity, security have become a major concern. In this context, Java plays an important role by integrating security in the underlying layers of the language. However, security is not always considered as an integral part of the software development process.

This thesis purposes to investigate security needs in general, and the specific techniques offered by the Java security architecture in particular. The original solutions of the language are emphasized, like the customized class loader, the access controller, the signing, sealing and guarding objects, and SSL.

Then this thesis describes a methodology based on UML, including current work done, for example UMLsec.

From this is derived a guide for implementing a secured Java project, that will show how to integrate security throughout the project lifecycle. This approach is illustrated by a practical example.

It is important to note that the main approach taken in this dissertation has been to reduce the complexity of the subject for a pedagogic purpose.

Keywords: security, Java, distributed systems, modelling, UML, UMLsec, software development

AVANT-PROPOS

Nous tenons à remercier toutes les personnes qui, d'une façon ou d'une autre, ont pu contribuer à la réalisation de ce mémoire.

Nous tenons particulièrement à remercier :

Notre promoteur, Monsieur Jean RAMAEKERS, pour sa disponibilité, son suivi efficace, son conseil, et pour toute l'attention qu'il a bien voulu nous accorder,

Monsieur Vincent ENGLEBERT, professeur aux Facultés Universitaires de Namur, pour son intérêt et ses encouragements,

Madame Marie d'UDEKEM-GEVERS, Conseillère à la formation pour la licence en informatique à horaire décalé, pour ses conseils et sa disponibilité,

Nos épouses respectives, Marisa et Marie, pour leur dévouement et leur patience,

Nos enfants respectifs, Alissa et Isabelle, Cynthia et Céline, pour l'accompagnement sonore et parfois musical avec lequel ils ont su entourer la réalisation de ce travail...

Joël MARCHAL,
Luc RENOTTE.

AVERTISSEMENT

Ce mémoire contient bon nombre de schémas et d'illustrations. Nous insistons sur le fait que la plupart de ces schémas ont été adaptés par rapport à la version originale dont nous nous sommes inspirés. C'est pourquoi, pour éviter d'alourdir inutilement le texte, nous n'avons pas jugé utile d'indiquer sur chaque figure le ou les documents d'origine.

Dans le même ordre d'idée, les références utilisées pour chaque section sont indiquées en tête de section. Les quelques citations ou références explicites sont toutefois répétées dans le texte.

Enfin, nous n'avons pas traduit systématiquement en français des termes anglais généralement admis par l'ensemble de la profession et souvent utilisés tels quels dans nos ouvrages de référence (ainsi par exemple du terme de *use case*). Nous avons donc, à l'occasion, privilégié les termes anglais reconnus plutôt que des traductions françaises parfois approximatives qui ne facilitent pas toujours la lecture et la compréhension générale.

TABLE DES MATIÈRES

TABLE DES FIGURES

GLOSSAIRE

ACL — *Access Control List*
Structure de données permettant de regrouper des permissions.

AES — *Advanced Encryption Standard*
Algorithme de chiffrement à clé symétrique, successeur de DES

API — *Application Programming Interface*
Interface de programmation sous forme de packages regroupant les fonctionnalités des classes de base du langage.

Applet — Programme Java intégré dans une page web (HTML), destinée à être exécutée dans un navigateur internet. L'applet est typiquement le composant mobile de Java, téléchargeable à la demande. Les règles de sécurité par défaut d'une applet sont beaucoup plus restrictives que pour une application Java.

Bytecode — Code Java obtenu après compilation ; ce code sera interprété par une machine virtuelle Java (c'est le « code machine » de la JVM). Le bytecode assure la portabilité d'un programme Java sur toute plateforme supportant une JVM.

CBC — *Cipher Block Chaining*
Mode de chiffrement DES : chiffrement par bloc (erreur sur 16 octets).

CERT/CC — *CERT Coordination Center*
Organisme centralisant les informations de sécurité sur Internet.

CFB — *Cipher FeedBack*
Mode de chiffrement DES : chiffrement par bloc (blocs inférieurs à 64bits).

ClassPath — Variable d'environnement Java indiquant le chemin de recherche des classes (utilisée par le chargeur de classe).

CORBA — *Common Object Request Broker Architecture*
Spécification de l'OMG normalisant les ORB pour garantir l'inter-opérabilité des applications conformes à cette norme.

COS Naming — *CORBA Object Naming Service*
Service de nommage pour les objets CORBA, accessible par l'API JNDI

CSP — *Cryptography Service Provider*
Fournisseur de Service Cryptographique implémenté dans JCA ; ce concept permet à tout fournisseur d'étendre les services de sécurité de JCA avec de nouvelles implémentations des fonctions de cryptographie.

DES — *Data Encryption Standard*
Algorithme de chiffrement symétrique standard (clés de 56 bits).

DESede — ou *3DES* ou *Triple DES*
Amélioration du DES (chiffrement en trois itérations avec 2 ou 3 clés).

Diffie-Hellman — Algorithme d'échange de clés dans lequel aucune des deux parties n'a besoin de connaître quoi que ce soit de l'autre (pas même sa clé publique).

DSA — *Digital Signature Algorithm*
Algorithme de chiffrement par clés asymétrique (souvent de 512 à 1024 bits).

ECB — *Electronic CodeBook*
Mode de chiffrement DES : chiffrement par bloc (blocs indépendants).

FIPS — *Federal Information Processing Standard*
Organisme de standardisation.

GIOP — *General Inter-ORB Protocol*
Principal protocole de communication de CORBA, basé sur TCP/IP.

HTTPS — *HyperText Transport Protocol Secure*
HTTP sécurisé basé sur SSL

IDL — *Interface Definition Language*
Langage de définition des objets CORBA. Gère les *skeletons* et les *stubs*.
Technologie d'objets distribués similaire à RMI mais basée sur CORBA.

IDS — *Intrusion Detection System*
Systèmes de détection d'intrusion, généralement utilisés derrière des firewalls.

IETF — *Internet Engineering Task Force*
Organisme notamment à l'origine du protocole TLS.

IIOP — *Internet Inter-ORB Protocol*
Protocole standard de CORBA basé sur TCP/IP, généralisé par GIOP. Assure la communication entre les ORB. Utilisable aussi sur RMI au lieu de JRMP.

Invocation API — API permettant à des applications natives (par exemple en C) de charger une JVM et d'exécuter du code Java. C'est la réciproque de JNI.

J2EE — *Java 2 Platform, Enterprise Edition*
Version entreprise de la plate-forme Java. Avant JDK 1.4, nécessaire pour obtenir les API de sécurité (JAAS, JSSE et JCE)

JAAS — *Java Authentication ans Authorization Service*
Package *javax.security.auth* fournissant les services d'authentification et de contrôle d'accès des utilisateurs. Intégré dans JDK 1.4.

JAR — *Java ARchive*
Format standard (souvent compressé) groupant les classes Java compilées et les ressources utiles au déploiement d'une application Java.

Java IDL — API Java regroupant les fonctionnalités permettant l'interopérabilité et l'intégration dans CORBA (voir IDL)

JCA — *Java Cryptography Architecture*
Package regroupant les fonctions de base de sécurité hors chiffrement. Implémente le concept de fournisseur de service cryptographique (cf. CSP).
Intégré dans JDK 1.4.
NOTE : Ne pas confondre avec *J2EE Connector Architecture* (gère l'accès aux applications hétérogènes d'un système).

JCE 1.2 — *Java Cryptography Extension*
Package regroupant les fonctions standard de chiffrement/déchiffrement, de génération des clés, et des MACs. Intégré dans JDK 1.4.

JDBC *Java DataBase Connectivity*
API normalisée pour l'accès aux bases de données relationnelles.

JDK *Java Development Kit*
Kit de développement Java incluant notamment le compilateur, la machine virtuelle Java et l'ensemble des classes de base du langage.

JKS *Java Key Store*
Implémentation du keystore de Java 2

JNDI *Java Naming and Directory Interface*
API définissant les accès aux services de nommage et aux services d'annuaires tels que *COS Naming* ou *RMI Registry*.

JNI *Java Native Interface*
Interface permettant à du code Java de faire appel à du code natif (comme des DLL sous Windows). Réciproque de *Invocation API*.

JRE *Java Runtime Engine*
Environnement d'exécution Java ; contient la JVM et les classes de base.

JRMP *Java Remote Method Protocol*
Protocole Java spécifique utilisé pour la communication dans RMI.

JSSE *Java Secure Socket Extension*
API de sécurité réseau, offrant les services SSL et TLS. Intégré dans JDK 1.4.

JVM *Java Virtual Machine*
Machine virtuelle Java, permettant d'exécuter les bytecodes. Une JVM est un programme natif qui simule un environnement d'exécution réel. C'est elle qui permet la portabilité des applications Java.

Kerberos Système d'authentification par mot de passe

keystore Un keystore (réserve de clés) est un conteneur de clés secrètes, de paires de clés publiques/privées et de certificats validant une clé publique.

MAC *Message Authentication Code*
Algorithme de condensé de messages utilisant une clé secrète.

MD5 *Message Digest version 5*
Algorithme de condensé de messages sur 128 bits.

Middleware Le middleware joue le rôle de pont de communication entre les différents modules distribués. C'est l'élément intermédiaire d'une architecture 3-tiers.

MOM *Message-Oriented Middleware*
Middleware permettant à des applications hétérogènes de communiquer à travers des files de messages (*« messages queuing »*).

NIST *National Institute of Standards and Technology*
Organisme de normalisation américain, notamment à l'origine de DSA.

OFB *Output FeedBack*
Mode de chiffrement DES : chiffrement par flux (données vocales).

OMG *Object Management Group*
Organisme de normalisation des technologies objets ; gère UML et CORBA.

ORB *Object Request Broker*
Couche logicielle offrant les services de communication normalisés pour l'interaction d'objets répartis (CORBA et IIOP).

Package — Ensemble de classes, regroupées généralement suivant leurs fonctionnalités. Par exemple le package standard *java.io* regroupe les classes d'entrées-sorties, *java.net* celles liées aux connexions réseaux, etc.

PCBC — *Propagating CBC*
Mode de chiffrement DES : chiffrement par bloc (propagation d'erreur).

PKCS#5 — *Public Key Cryptography Standard* version 5
Technique de *padding* utilisé dans le DES.

Principal — En sécurité, le mot *principal* est utilisé pour désigner un individu, une organisation ou un autre émetteur ou récepteur de messages. Dans un système sécurisé, tout principal doit être identifié de manière univoque.

RC5 — *Ron'sCode* version 5 (Ronald Rivest)
Algorithme de chiffrement à clé secrète par blocs (plusieurs tailles).

RMI — *Remote Method Invocation*
Technologie standard Java pour l'utilisation d'objets distants. RMI peut être vu comme un ORB dédicacé Java. Il fonctionne avec une Registry, des stubs et skeletons, et avec un protocole JRMP ou IIOP.

RMI Registry — Service de désignation d'objets (annuaire) de RMI, utilisé pour rechercher la référence à un objet distant.

RMI-IIOP — Version de RMI qui utilise le protocole de communication IIOP de CORBA pour l'échange de données entre objets distants. IIOP est le nouveau standard Sun depuis JDK 1.3, à la place de JRMP. Il offre comme avantage d'ouvrir la communication avec tout objet implémentant IIOP. RMI-IIOP recouvre les technologies Java RMI et Java IDL.

RMI-JRMP — Version de RMI qui utilise le protocole de communication JRMP pour l'échange de données entre objets distants. Depuis JDK 1.3, IIOP est recommandé à la place de JRMP.

RPC — *Remote Procedure Call*
Technologie permettant à un objet de déclencher l'exécution d'une procédure sur une machine distante.

RSA — *Rivest, Shamir, Adleman*
Algorithme de chiffrement à clés asymétriques (longueur de clés variable).

Sandbox — Le « bac à sable » dans lequel sont confinées les applets sur le poste client. Il s'agit d'un espace de confinement destiné à limiter l'accès aux ressources.

Sérialisation — Mécanisme standard permettant de transformer un objet en un flux d'octets, pour le reconstituer ultérieurement (par exemple après transfert réseau).

SHA-1 — *Secure Hash Algorithm version 1*
Algorithme de condensé de messages sur 160 bits.

Skeleton — Composant logiciel chargé de la communication entre objets distants (RMI, CORBA…) ; un *skeleton* côté serveur communique avec un *stub* côté client.

SKIP — *Simple Key Management for Internet Protocol*
Standard Internet normalisant le protocole d'échange de clé Diffie-Hellman.

SPI — *Service Provider Interface*
Interface Java de fournisseur de service (à implémenter par un CSP).

SSL — *Secure Socket Layer*
Couche de communication sécurisée basée sur les sockets.

Stub Composant logiciel chargé de la communication entre objets distants (RMI, CORBA...) ; un *stub* côté client communique avec un *skeleton* côté serveur.

Synchronized Associé à une méthode, ce qualificatif permet d'obtenir l'accès exclusif à l'objet pour toute autre méthode synchronisée. Il s'agit en quelque sorte d'un sémaphore d'utilisation de l'objet. Ce mécanisme est utilisé pour la coordination entre les threads et l'accès aux ressources.

TCP/IP *Transmission Control Protocol / Internet Protocol*
Protocole standard de communication réseau d'entreprises et d'Internet.

TLS *Transport Layer Security*
Protocole élaboré par l'IETF; amélioration de SSL*v*3.

UML *Unified Modeling Language*
Norme de l'OMG pour la modélisation d'applications orientées objets.

URL *Uniform Ressource Locator*
Syntaxe normalisée de la localisation d'une ressource réseau.

WTLS *Wireless TLS*
TLS version sans fil.

1. INTRODUCTION

La motivation de ce mémoire est née d'un double intérêt de notre part : l'intérêt quant aux technologies orientées objets en général et à Java en particulier, et l'intérêt pour les technologies de sécurité. Il faut souligner ici que, hormis les notions académiques assimilées durant les cours de licence en informatique, nous n'avions pas de connaissances particulières dans ces domaines techniques.

La principale motivation de ce mémoire consistait à investir ces techniques pour en comprendre les fondements, les mécanismes, et les implications. Il s'agit donc avant tout d'un **mémoire d'investigation**, à caractère prioritairement académique et pédagogique. Relevons à ce propos, que certaines parties de ce mémoire seront utilisées dans le cadre du cours « Sécurité des Systèmes Informatiques » du Professeur Jean Ramaekers.

Préalablement à la présentation des buts que nous avons poursuivis, il nous paraît utile de souligner un paramètre important de notre travail, qui est la quantité considérable d'informations qu'il est nécessaire d'intégrer pour arriver à dégager une vue suffisamment claire des technologies traitées.

En effet, ces technologies étant en permanente et rapide mutation, les documents les plus à même de nous renseigner sur les évolutions récentes de ces techniques sont souvent très épars et fragmentaires. De plus, ils font régulièrement émerger davantage de nouvelles questions qu'ils n'en résolvent. L'explosion des sources qui en résulte immanquablement élève le facteur documentation au rôle de consommateur essentiel du temps de travail.

Il est en outre apparu à plusieurs reprises que les conclusions que nous tirions d'une étape particulière de recherche se trouvaient confirmées par de nouvelles découvertes ultérieures.

Or, ce temps de recherche et cette dynamique de préparation ne sont pas immédiatement apparents lors de la présentation des résultats de ces investigations ; il nous semblait donc nécessaire d'attirer l'attention du lecteur sur ce facteur clé.

OBJECTIFS PHARES

Comme nous venons de le suggérer, ce mémoire ne se propose pas d'atteindre des objectifs techniques entièrement définis dès le départ. A la place, nous nous proposons de fixer des **objectifs-phares**, comme indicateur de direction à suivre et comme élément discriminant à l'heure des divers choix qu'il faudra immanquablement opérer au fil de notre travail.

Ces objectifs-phares sont les suivants :
1) *Etudier les techniques* de sécurité liées à l'environnement *Java* ;
2) Fournir une *description pédagogique* des techniques étudiées ; c'est à dire permettre à un lecteur non initié à ces techniques d'appréhender la complexité de la sécurité liée à un projet de type Java, sans devoir lui-même étudier une masse de documentation rédhibitoire ;
3) Etudier les aspects de la sécurité *dans le cadre d'applications distribuées*, principal centre d'intérêt des technologies de sécurité dans un environnement du type Java ;

4) Faire émerger les *lignes de conduite de la sécurisation d'un projet Java*, et l'intégration de la sécurité dans le processus de développement d'une application, de la modélisation au déploiement.

ORGANISATION DU MÉMOIRE

Le mémoire peut être scindé en trois grandes parties :

- une partie de description technique, permettant de présenter le domaine de travail et le résultat de nos investigations ;
- une partie orientée méthodologie, dans laquelle nous proposerons un guide ;
- une partie orientée application, au travers de laquelle nous illustrerons les deux premières parties.

PARTIE I : CONCEPTS DE SÉCURITÉ ET TECHNIQUES JAVA

Nous introduirons tout d'abord les principales techniques de sécurité de façon générique, c'est-à-dire les principales préoccupations de sécurité liées à la distribution d'applications au travers d'u n réseau. Nous présenterons donc les principes des services de sécurité, ainsi que les différents types d'attaques auxquelles les systèmes sont confrontés.

Nous nous attarderons ensuite brièvement sur les principes de la programmation objets et sur les implications particulières de la sécurité dans les langages de ce type.

Ensuite, nous présenterons la panoplie des technologies de sécurité mises en œuvre dans l'environnement Java : quelle est l'architecture de sécurité de Java, comment sont couverts les divers services de sécurité, et quels sont les mécanismes originaux proposés par ce langage en particulier.

L'implémentation de la sécurité étant particulièrement pertinente dans le cadre d'applications réparties, nous nous intéresserons aux principes des architectures distribuées, en nous basant essentiellement sur les architectures CORBA et Java-RMI. Nous proposerons également un comparatif de ces technologies, et nous verrons comment leur évolution mène à leur convergence.

PARTIE II : CONCEPTION ET MODÉLISATION D'APPLICATIONS SÉCURISÉES

Nous essayerons de mettre au jour une façon de modéliser les informations de sécurité dans le cadre d'outils de modélisation de projet Java. Nous nous baserons pour cela sur le standard UML, et sur une extension appelée UMLsec.

Ensuite, nous tenterons de dégager les pistes essentielles pour élaborer un guide de la sécurisation d'un projet Java en montrant l'importance d'intégrer la sécurité dès la première phase de modélisation.

PARTIE III : EXEMPLE D'APPLICATION

Dans cette troisième et dernière partie, nous illustrerons les principes, les techniques et le guide proposés dans les deux premières parties, en les mettant en œuvre dans le cadre d'une application illustrative basée sur un environnement de type accès bancaire à distance. Nous tenterons de voir, à travers cette application exemple, comment exploiter les principes de modélisation et de sécurisation exposés dans notre guide.

PARTIE I :

CONCEPTS DE SÉCURITÉ ET TECHNIQUES JAVA

Dans cette première partie, nous nous intéressons à la sécurité des Systèmes d'Information. Nous décrivons les exigences de sécurité, les principaux types d'incidents ou d'attaques, les principaux protocoles standards, et les problèmes particuliers liés à la P.O.O.

Ensuite, nous investigons les techniques mises en œuvre dans le langage Java en matière de sécurité : architecture de sécurité, extensions du langage, techniques originales de sécurité. Certaines techniques jugées particulièrement intéressantes sont décrites en détails : chargeur de classe, contrôleur d'accès, objets signés/scellés/gardés, et protocole SSL.

Afin de mieux cerner le contexte de ces applications, nous décrivons ensuite les principales caractéristiques des architectures distribuées, notamment CORBA et RMI.

2. SÉCURITÉ DE L'INFORMATION

2.1. SERVICES DE SÉCURITÉ

La sécurité est un sujet très vaste, tant par les aspects qu'elle aborde que par les domaines d'implication. Dans le cadre de ce travail, nous nous intéressons plus particulièrement à la sécurité des réseaux informatiques, du réseau local au réseau public mondial Internet.

Nous décrivons pour commencer un modèle de base de sécurité, sous forme de *services* de protection qu'un fournisseur de services de sécurité doit pouvoir offrir à un fournisseur de ressources, afin que ce dernier puisse mettre des données ou des applications à disposition d'un client de façon sécurisée.

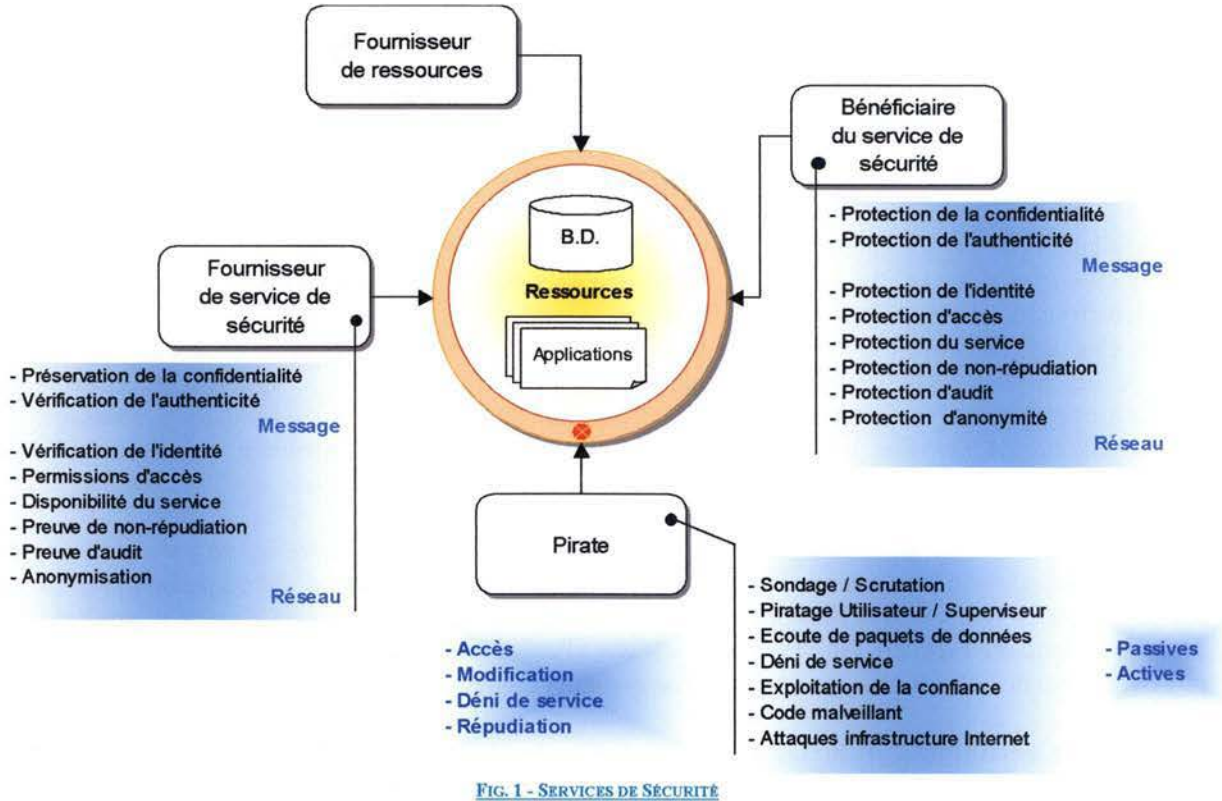


FIG. 1 - SERVICES DE SÉCURITÉ

Ces services de sécurité doivent s'opposer aux efforts de tout pirate désireux de falsifier les ressources, d'y accéder, de les remplacer, ou d'en perturber l'accès.

Les premiers services concernent les messages. Par *message*, nous entendons toute requête ou réponse à une requête. Ces services essentiels concernent la vérification de l'authenticité et la préservation de la confidentialité des messages.

Les services présentés ensuite concernent l'utilisation d'une ressource par l'intermédiaire d'un réseau : disponibilité du service, vérification de l'identité, permissions d'accès, preuve de non-répudiation et preuve d'audit.

Enfin, nous abordons brièvement un problème plus particulier et méconnu, celui de l'anonymisation.

2.1.1. PRÉSERVATION DE LA CONFIDENTIALITÉ

[JAWORSKI 01] • [CERT 97]

Le bénéficiaire dispose de certaines garanties sur la préservation de la confidentialité des données ou de l'application.

Les données sont chiffrées par le fournisseur de ressources à l'aide d'un moteur cryptographique : c'est l'opération de **chiffrement**. Ces informations chiffrées peuvent être envoyées sur un réseau de communication avec la garantie que la confidentialité des données est préservée.

Chez le bénéficiaire, un processus inverse peut alors traiter les données dans un moteur cryptographique identique, et convertir les données chiffrées en leur forme d'origine (en clair) : c'est l'opération de **déchiffrement**. Seuls les moteurs cryptographiques possédant la clé secrète adéquate et connaissant l'algorithme à exécuter peuvent effectuer le traitement cryptographique inverse.

FIG. 2 – CHIFFREMENT / DÉCHIFFREMENT

On distinguera deux types de clés dans les opérations de chiffrement/déchiffrement :

- les **clés symétriques** : la même clé secrète est utilisée pour les processus de chiffrement et de déchiffrement (la clé doit être échangée de façon confidentielle).
- les **clés asymétriques** : les processus de chiffrement et déchiffrement utilisent des clés différentes et complémentaires. Le principal dispose donc d'une paire de clés : l'une privée (et secrète), l'autre publique (et diffusable librement). Pour garantir la confidentialité du message, le fournisseur de données utilise la clé publique du bénéficiaire (distribuée librement) pour le chiffrement, tandis que le bénéficiaire utilise sa clé privée (et secrète) pour le déchiffrement.

Lorsqu'une information peut être lue ou copiée par une personne non autorisée, on parle de perte de confidentialité (*loss of confidentiality*). Pour certains types d'information, la confidentialité est un attribut essentiel : par exemple, les données de recherches, les spécifications de nouveaux produits, les données à caractère médical, etc. Dans certains cas, il existe même une obligation légale à protéger le caractère privé de ces données (cas typique des hôpitaux ou des cabinets de médecins).

Principaux algorithmes : DES (Data Encryption Standard), TripleDES, RC2 et RC4, RC5 pour les clés symétriques ; RSA (Rivest Shamir Adleman), Diffie-Hellman pour les clés asymétriques.

2.1.2. VÉRIFICATION DE L'AUTHENTICITÉ

[JAWORSKI 01] • [CERT 97]

Le bénéficiaire peut vérifier que les données qu'il a reçues sont exactes et intactes (la corruption des données n'est pas forcément volontaire) et que leur origine est garantie.

L'authenticité des données recouvre deux aspects fondamentaux :
- l'**intégrité** du contenu ;
- la garantie de l'**origine** des données.

Pour préserver l'intégrité d'une suite de données, on lui adjoint un ***digest de message***. Le digest de message est un type de fonction particulier, dite unidirectionnelle (ou fonction de hachage).

Une fonction unidirectionnelle est facile à calculer mais très difficile à inverser. Elle possède les propriétés suivantes :
- *Non-réversibilité* : étant donné une valeur de digest de message, il n'est pas possible de calculer un message qui produira cette valeur en sortie de la fonction de digest de message.
- *Unicité* : il est très hautement improbable de trouver deux messages produisant la même valeur de digest de message.

Le bénéficiaire reçoit le message et le digest de message. Au départ du message reçu, il recalcule le digest. L'intégrité des données est vérifiée si le digest calculé est identique au digest reçu.

Lorsque l'on s'aperçoit que l'information a été modifiée (les deux digests ne correspondent pas), on parle de perte d'intégrité (*loss of integrity*). Cela signifie qu'un changement non autorisé a été opéré, soit en raison d'une erreur involontaire, soit de manière intentionnelle.

L'intégrité peut être particulièrement critique à assurer, notamment pour des données relevant de la sécurité ou à caractère financier; c'est le cas par exemple du contrôle du trafic aérien ou de la gestion des comptes bancaires.

Principaux algorithmes : MD5 (Message Digest 5), SHA-1 (Secure Hash Algorithm) et MAC (Message Authentication Code).

Les digests de message sont très pratiques pour indiquer qu'un message ou un autre objet a été, accidentellement ou délibérément, altéré, mais ils ne peuvent pas garantir l'origine des données.

Dès lors, pour garantir l'*origine* des données, il faut y adjoindre une **signature**. Une signature peut être générée en appliquant une clé privée sur le digest de message. Ce digest de message pourra être déchiffré à l'aide de la clé publique du signataire, ce qui garantit donc son identité.

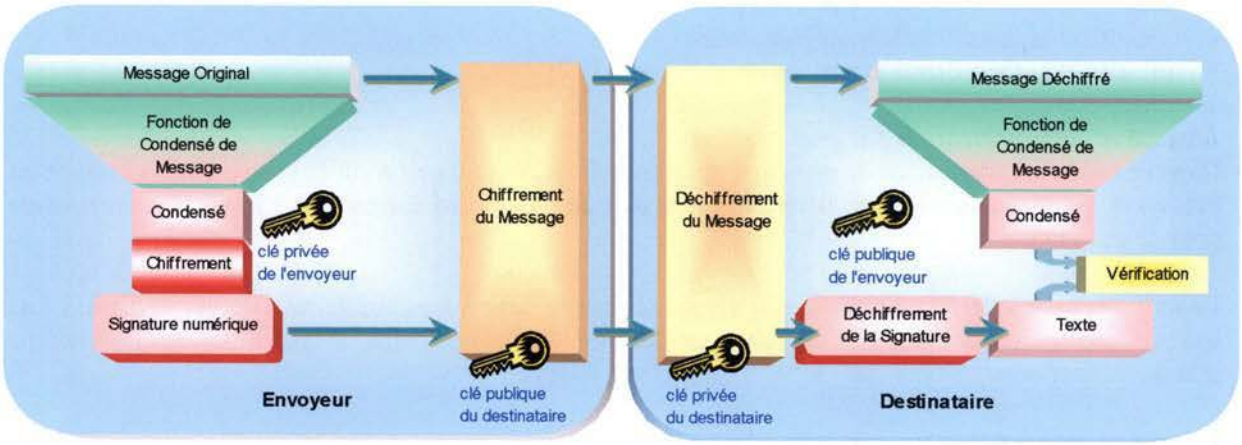


FIG. 3 – SIGNATURE NUMÉRIQUE

Notons au passage que lorsqu'un digest de message ainsi déchiffré ne correspond pas au digest généré au départ du message original, il n'est pas possible de déterminer si c'est le contenu du message ou son origine qui n'est pas vérifiée. Mais dans les deux cas, le message doit être refusé.

2.1.3. VÉRIFICATION DE L'IDENTITÉ (*AUTHENTICATION*)

[JAWORSKI 01]

> La vérification de l'identité d'un principal permet au bénéficiaire de vérifier que l'identité associée à la source des données ou de l'application est bien celle indiquée.

Cette vérification d'identité permet aussi de s'assurer que les permissions d'accès aux ressources ont été correctement authentifiées, c'est-à-dire qu'avant d'accorder l'accès à une ressource, on a déterminé l'identité du *principal* demandant cet accès.

Dans les systèmes de sécurité, le mot *principal* est utilisé pour désigner un individu, une organisation ou un autre émetteur ou récepteur de messages. Dans un système sécurisé, tout principal doit être identifié de manière univoque.

L'identification d'un principal doit s'effectuer de manière sécurisée : il ne suffit pas d'indiquer son identité. S'identifier comme un principal ayant plus de privilèges que ceux que l'on a vraiment pourraient permettre d'accéder à certaines ressources normalement inaccessibles. **L'affectation de l'identité du principal dans un système est donc une opération essentielle pour la sécurité**.

L'authentification est le moyen par lequel un principal s'identifie sur un système. A cette fin, il doit généralement fournir au système des informations secrètes qu'il est sensé être seul à connaître ou être capable de générer. Lorsque le principal est correctement identifié, le système d'authentification lui associe un ensemble de références définissant les droits qui lui sont attribués.

Ces références peuvent être *déléguées* à d'autres objets par le principal afin que ces autres objets soient capables d'effectuer des opérations en son nom. Par exemple, supposons qu'un client, préalablement authentifié, appelle un serveur de requête et qu'il lui transmette ses références ; le serveur appellera ensuite un objet carte de crédit à qui il transmettra les références du client pour que l'opération puisse être effectuée pour son compte. Sans délégation, l'objet carte de crédit verrait tous les appels effectués sous l'identité du serveur de requêtes.

Diverses techniques d'authentification sont apparues avec le temps, parmi lesquelles :
- identité et authentification par mot de passe ;
- identité et authentification par jeton physique ;
- identité et authentification par biométrie ;
- identité et authentification par certificat.

Identité et authentification par mot de passe
Dans ce mode, un identifiant de principal et un mot de passe sont saisis sur le système et transmis à un processus d'authentification qui détermine si le mot de passe saisi correspond à la version enregistrée pour ce principal.

Le mot de passe fonctionne comme une clé secrète, avec des risques similaires : nécessité d'avoir une banque des mots de passe et d'échanger le mot de passe entre le principal et le système d'authentification.

Identité et authentification par jeton physique
Dans ce cas, l'authentification d'un principal implique un élément physique comme une carte de distributeur automatique. Les cartes à puces contiennent des microprocesseurs incorporés permettant d'offrir des moyens plus configurables de contrôle d'identité par jeton physique.

Identité et authentification par biométrie

La biométrie implique l'utilisation d'un élément physique de la personne (par exemple, l'empreinte digitale ou les caractéristiques rétiniennes d'un œil). Cette technique a pour avantage la difficulté pour un intrus d'obtenir la clé secrète et la grande facilité pour le principal à conserver sa clé. Néanmoins, on retrouve ici les même inconvénients que pour l'authentification par mot de passe (nécessité d'avoir une banque des clés secrètes et d'échanger les clés), avec une difficulté supplémentaire : l'impossibilité de changer de clé en cas de vol durant la transmission ou de pillage de la banque des clés.

Identité et authentification par certificat

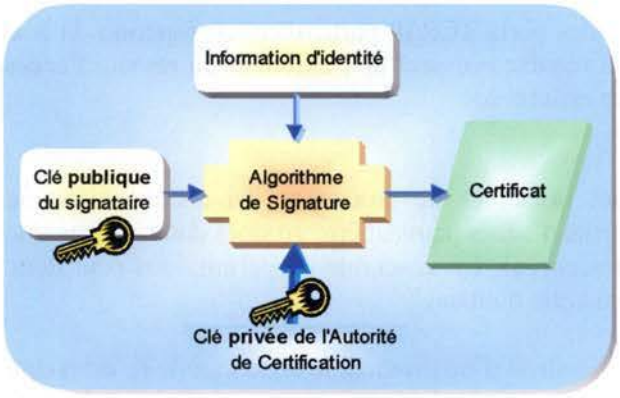


FIG. 4 – CERTIFICAT NUMÉRIQUE

Un certificat est un bloc de données contenant les informations qui permettent d'identifier un principal : sa clé publique, son identité, la validité du certificat, des informations sur l'émetteur du certificat et une signature générée par cet émetteur. Un certificat est signé par une *autorité de certification* tierce, dont la clé publique est elle-même certifiée par une autorité de certification telle que Verisign dont le certificat est pré-configuré dans la plupart des navigateurs Web.

En outre, les certificats peuvent se chaîner : ainsi un certificat C-3, certifié par C-2, lui-même certifié par C-1 pourra être accepté par un destinataire reconnaissant C-1. C-1 pourrait être, par exemple, Verisign, déjà mentionné.

Principaux algorithmes : PKCS (*Public Key Cryptography Standards*) définit un format binaire utilisable pour l'enregistrement des certificats : PKCS#1, PKCS#2, ... On rencontre aussi le format ASCI PEM (*Privacy Enhanced Mail*).

2.1.4. PERMISSION D'ACCÈS (*AUTHORIZATION*)

[JAWORSKI 01] • [JURJENS 01] • [OAKS 01]

> L'accès aux données, applications ou autres ressources essentielles a été fourni avec les permissions correctes.

Il s'agit donc de fournir un contrôle d'accès aux utilisateurs des ressources, les *principaux*, en fonction de leur identité, du groupe auquel ils appartiennent, de leur rôle, ou encore du niveau d'accès auquel ils ont droit. Par ailleurs, le contrôle d'accès aux ressources peut aussi être fonction du lieu d'origine de la requête (depuis son bureau via LAN ou son domicile via WAN).

Enfin, le contrôle d'accès doit répondre aux deux spécifications suivantes :

- il doit être suffisamment solide pour empêcher toute personne non autorisée d'accéder aux ressources sensibles ;
- il ne doit pas être trop restrictif et supprimer l'accès légitime aux principaux autorisés.

Contrôle d'accès discrétionnaire

Ce type de contrôle d'accès se base sur l'identité des principaux (individus ou groupes) et implique habituellement de gérer une liste de contrôle d'accès (ACL, Acces Control List).

Contrôle d'accès selon le rôle
Le contrôle d'accès par rôle exige de gérer une liste de rôles et d'établir des correspondances entre le rôle et l'utilisateur ou le groupe d'utilisateurs.

Contrôle d'accès obligatoire
Ce contrôle se fonde sur un niveau de classification assumé par le principal et indiquant son degré d'approbation (par exemple, niveau élevé pour degré top secret, moyen pour confidentiel et faible pour non classifié). Un niveau de classification est aussi associé à une ressource indiquant le degré minimal de classification nécessaire pour qu'un principal y accède.

Contrôle d'accès par pare-feu
Un pare-feu est un mécanisme qui limite l'accès à des ports TCP/IP particuliers en fonction du lieu d'origine de la requête de connexion entrante : si la requête provient de l'extérieur du réseau, l'accès peut être restreint ; si elle provient du réseau, l'accès est accordé.

Domaines
Le contrôle d'accès peut aussi se définir par des domaines de protection. Divers machines ou traitement possédant tous la même stratégie de contrôle d'accès peuvent être groupés dans un domaine de sécurité (ou *realm*, ou domaine de stratégie de sécurité). Un ensemble de permissions peut donc être accordé à un domaine, et un autre ensemble à un autre domaine.

Un domaine de sécurité peut aussi faire référence à l'endroit d'où provient le code mobile (c'est-à-dire une base de code). Ainsi il est possible de définir un ensemble de permissions à accorder à du code mobile provenant d'une URL donnée.

Le code d'un domaine peut approuver l'appel de certaines opérations par du code du même domaine, mais ne pas l'accepter pour du code résidant hors du domaine.

D'un point de vue hiérarchique, les domaines peuvent être découpés en sous-domaines ou fédérés. Les sous-domaines héritent des permissions accordées au domaine englobant : ces permissions peuvent être réduites au niveau du sous-domaine ; elles sont rarement élargies. Les fédérations de domaines accordent un accès mutuel aux ressources au niveau du domaine.

2.1.5. DISPONIBILITÉ DU SERVICE

[JAWORSKI 01] ♦ [CERT 97]

Le bénéficiaire peut accéder aux données, aux applications ou à d'autres ressources quand il le souhaite.

Cette disponibilité du service recouvre deux aspects :
- les ressources doivent être physiquement accessibles (les réseaux, serveurs dont elles dépendent doivent être en fonctionnement), ce qui implique généralement une réplication de ces ressources (systèmes distribués) ;
- tout principal possédant un droit sur une ressource doit pouvoir en user à tout moment, c'est-à-dire que le système de sécurité ne doit pas être trop restrictif et empêcher l'accès à une personne autorisée.

La disponibilité est souvent l'attribut le plus important des services orientés entreprises dont le travail repose sur l'information (compagnies aériennes, gestion de stock en-ligne). La disponibilité du réseau lui-même est importante pour tout qui est relié à son travail via un réseau. Lorsqu'un utilisateur ne peut accéder à un service spécifique sur un réseau, on parle de déni de service (*denial of service*).

2.1.6. PREUVE DE NON-RÉPUDIATION

[JAWORSKI 01]

Le bénéficiaire possède une preuve concernant l'origine des données ou de l'application, ou une preuve qu'un destinataire a effectivement reçu les données qui lui ont été envoyées.

Cette preuve est fournie par l'émission d'un jeton de non-répudiation (NR) en même temps que l'action d'envoi ou de réception des données. Ce jeton doit associer de manière indubitable l'identité du principal qui envoie ou reçoit les données et les données elle-mêmes. Cela s'effectue habituellement en chiffrant un digest des données à l'aide de la clé privée du principal.

Quand un autre principal obtient un jeton NR ainsi généré, il a une preuve que l'action s'est produite et a été exécutée par ce principal. En cas de conflit ultérieur, cette preuve peut être présentée à un arbitre ou à une autorité de non-répudiation.

2.1.7. PREUVE D'AUDIT

[JAWORSKI 01]

Le bénéficiaire peut prouver que des opérations essentielles pour la sécurité se sont produites sur le système.

L'audit implique la **journalisation** d'opérations cruciales pour la sécurité se produisant dans un domaine de sécurité. L'audit de ces opérations peut être utile pour tenter de pister la suite d'évènements qui ont conduit à une fraude, ou éventuellement pour déterminer l'identité du pirate. L'audit s'effectue habituellement durant l'authentification (ouverture de session), la fin de session, l'accès aux ressources critiques, et toute modification des propriétés de sécurité d'un système.

A chacune de ces opérations, on enregistre les informations suivantes : nature de l'opération, horodatage, identité du principal qui a déclenché l'événement, identification de la ressource, permissions demandées, origine de la requête,...

Les journaux d'audit sont des éléments sensibles pour la sécurité et doivent être protégés contre la falsification ou la destruction. Il sont éventuellement chiffrés pour ajouter de la confidentialité.

L'audit peut en quelque sorte être considéré comme une version *faible* de la non-répudiation : la plupart des opérations sont enregistrées, mais certaines ne le sont pas ; en cas d'usurpation d'identité, c'est l'identité usurpée qui est enregistrée ; un administrateur système peut falsifier les journaux d'audit. Pour ces raisons, l'audit peut constituer une présomption mais non une preuve au sens juridique du terme.

2.1.8. ANONYMISATION

Les fichiers contenant des données confidentielles sur les individus doivent répondre à des règles strictes d'anonymat.

Ainsi en est-il, par exemple, des données médicales dans les hôpitaux ou des données de l'Institut National des Statistiques. Néanmoins, certaines personnes sont autorisées à consulter ces données dans le cadre de leur travail, notamment pour construire des statistiques,.

Différents aspects peuvent être relevés :

- la détection des indiscrétions : comment faire la différence entre le médecin qui consulte les données et de son patient et le même médecin qui jette un regard indiscret sur le dossier médical d'un membre de sa famille ; l'employé de banque qui consulte le compte de son client ou celui de son collègue…
- la possibilité d'étudier certaines corrélations d'un point de vue statistique : par exemple pouvoir comparer l'évolution de certaines maladies sur une longue période implique de pouvoir regrouper par individu des données anonymes, reçues à différents moments (exemple de la banque Carrefour cité par J. RAMAEKERS).

Comment protéger ce type de données de l'indiscrétion ? D'une part l'accès aux données doit être accordé aux personnes qui en ont besoin pour des raisons professionnelles et d'autre part il devrait être refusé dans le cadre d'un intérêt privé. Le seul moyen de contrer ce type d'attaque est une vérification à posteriori : par exemple, en examinant les fichiers d'audit du système.

2.2. INCIDENTS DE SÉCURITÉ

[CERT 97] ◆ [MAIWALD 01] ◆ [ZIFF 02]

2.2.1. RAPPEL HISTORIQUE SUR INTERNET

Lors de sa conception, le réseau ARPANET, l'ancêtre d'Internet, avait surtout été conçu pour résister aux attaques de type 'déni de service' : le but principal (et original pour l'époque) était de créer un réseau capable de continuer à fonctionner en cas de panne d'une ou plusieurs de ses sections. Cette résistance aux pannes de réseau reposait sur le routage, c'est-à-dire la possibilité de rediriger toute information par un autre chemin en cas de panne d'une section du réseau.

Les autres caractéristiques du protocole ARPANET étaient l'ouverture et la flexibilité mais pas la sécurité. En effet, ce réseau était réservé à l'armée américaine et aux universités et il n'était pas prévu de l'ouvrir à un large public. Il s'agissait d'un petit groupe de personnes qui se connaissaient et qui se faisaient mutuellement confiance.

Aujourd'hui, Internet est un réseau public mondial où peuvent se connecter des millions de personnes, s'échanger des données confidentielles, s'effectuer des transactions commerciales et des paiements par cartes de crédit. Dès lors, la sécurité y est devenue un besoin essentiel.

2.2.2. SOURCES D'INCIDENTS DE SÉCURITÉ SUR UN RÉSEAU

Un incident de sécurité sur un réseau (*network security incident*) peut être n'importe quelle activité relative au réseau avec des implications négatives quant à la sécurité. Cela signifie généralement que cette activité viole une règle de sécurité explicite ou implicite du réseau.

A l'origine de ces incidents, on trouve le *hacker* (pirate) : celui-ci dispose d'une compétence technique non négligeable grâce à laquelle il est capable de se glisser dans la moindre faille de sécurité d'un système. On distingue les *black hats* et les *white hats* en fonction de leur volonté de nuire ou non, et, depuis peu, les *grey hats*, engagés pour la défense, en cas de guerre digitale.

Les motivations des pirates sont multiples ; selon les cas on criera au génie de l'informatique ou au criminel digital :

- recherche de l'exploit, challenge intellectuel, sentiment de puissance : pour prouver la vulnérabilité d'un site réputé inviolable ;

- vengeance : par exemple un employé licencié qui pirate son ancien employeur ;
- profit : pirater pour obtenir de l'argent, par exemple en interceptant une transaction de paiement à son profit ;
- curiosité : pour voir jusqu'où il est possible de pénétrer certains systèmes sans être détecté ;
- motivation politique : cela peut être au nom des droits de l'homme ou des animaux, de la liberté d'expression, etc.

Il existe pourtant une caractéristique commune à la communauté des pirates : la communication. Les pirates possèdent des newsgroups et publient des ouvrages sur les dernières techniques d'intrusion ; ils organisent des conférences sur le sujet. Ils identifient et montrent du doigt les systèmes mal sécurisés.

Un article récent publié par *Ziff Davis Smart Business* [ZIFF 02] dénombrait plus de 50 outils de *hacking* disponibles sur Internet. Citons par exemple les outils *datadictionary* capables de retrouver les mots de passe en testant plus d'un milliard de possibilités en 13 secondes. Un mot de passe de quatre caractères est ainsi *cracké* en 30 secondes.

Grâce à ce partage de connaissances et grâce à des outils logiciels faciles à utiliser, le succès des pirates augmente chaque jour, obligeant les responsables de sécurité à investir toujours plus dans la sécurité.

2.2.3. TYPES D'INCIDENTS / ATTAQUES

La liste des incidents et attaques ci-dessous est essentiellement basée sur un document du *CERT® Coordination Center*. Le CERT®/CC centralise les informations concernant la sécurité sur Internet.

- **Sondage *(Probe)***
  Ce type d'incident est caractérisé par une tentative inhabituelle pour accéder à un système ou pour découvrir une information à propos d'un système. Un exemple courant est d'essayer de s'identifier dans un système au moyen d'un compte inutilisé. C'est le principe du voleur qui essaye les poignées de porte pour entrer quelque part.

- **Scrutation *(Scan)***
  Il s'agit d'un grand nombre de sondages réalisés à l'aide d'un outil automatisé. C'est la recherche systématique du point faible du système.

- **Piratage d'un compte utilisateur *(Account Compromise)***
  C'est l'utilisation non autorisée d'un compte par quelqu'un d'autre que son propriétaire. Dans ce premier cas, il s'agit d'un compte utilisateur, ce qui limite les dégâts, mais il n'empêche que le propriétaire du compte s'expose à des pertes ou des vols de données. De plus, bien qu'il s'agisse d'un compte normal, il peut devenir le point d'entrée vers un plus grand accès au système.

- **Piratage d'un compte superviseur *(Root Compromise)***
  Le piratage d'un compte superviseur est semblable à l'incident précédent, excepté le fait que ce compte possède des privilèges spéciaux sur le système, ceux de « super utilisateur ». Le pirate qui parvient à se connecter en superviseur peut faire absolument ce qu'il veut : faire tourner ses propres programmes, changer la manière dont le système fonctionne et même cacher les traces de son intrusion.

- **Écoute des paquets de données *(Packet Sniffer)***
  Un renifleur de paquets permet de capturer les paquets d'informations lorsqu'ils voyagent sur le réseau. Ces informations peuvent inclure noms, mots de passe et informations confidentielles et voyager « en clair » sur le réseau. A l'aide des mots de passe ainsi capturés, un pirate peut se livrer

à des attaques plus larges sur le système. Notons aussi que l'installation d'un renifleur ne demande pas nécessairement un accès privilégié. Citons à ce sujet les résultats d'une récente enquête de l'hebdomadaire Data News relative aux réseaux sans fil : la plupart des réseaux testés à l'occasion de cette enquête diffusaient les paquets en clair dans la zone de réception dudit réseau, notamment aux abords de l'entreprise, permettant ainsi à l'enquêteur muni d'un portable de se connecter sans difficulté.

- **Déni de Service** ***(Denial of Service)***
  Le but de cet incident est différent des incidents précédemment décrits : il ne s'agit plus d'obtenir un accès non autorisé à des données mais bien d'empêcher les utilisateurs légitimes d'utiliser le service auquel ils sont connectés. Ce type d'attaque peut prendre différentes formes : les pirates peuvent inonder un réseau avec de grands volumes de données de sorte que le système se bloque ou que les temps d'attente fassent perdre les connexions au réseau ; ils peuvent aussi interrompre physiquement le réseau ou déformer les données en transit, même si elles sont chiffrées.

- **Exploitation de la confiance** ***(Exploitation of Trust)***
  Les ordinateurs ont entre eux une relation de confiance mutuelle. Par exemple, avant d'exécuter certaines commandes, l'ordinateur vérifie une série de fichiers qui spécifie quels sont les autres ordinateurs du réseau autorisés à utiliser ces commandes. Si le pirate se fait passer pour un de ces ordinateurs de confiance, il pourra obtenir un accès non autorisé aux autres ordinateurs.

- **Code Malveillant** ***(Malicious Code)***
  Code malveillant est un terme générique pour désigner les programmes qui, lorsqu'ils sont exécutés, causent des résultats non désirés sur un système. En général, les utilisateurs du système ne se méfient pas de ce type de programme jusqu'à ce qu'ils découvrent les dégâts. Il s'agit des chevaux de Troie, des virus et des worms (vers). Les chevaux de Troie et les virus sont généralement cachés dans un programme légitime ou dans des fichiers altérés par le pirate. Les worms se multiplient automatiquement sans intervention humaine après leur démarrage. Les virus sont aussi des programmes qui s'automultiplient mais ils demandent souvent une intervention supplémentaire. Tous ces programmes peuvent causer des pertes de données, des retards, des dénis de services et autres types d'incident.

- **Attaques de l'Infrastructure d'Internet** ***(Internet Infrastructure Attacks)***
  Il y a aussi de rares mais sérieuses attaques de l'infrastructure même d'Internet : des composants comme les serveurs de noms (DNS), les accès des providers, les grands sites d'archives dont dépendent beaucoup de personnes.

2.2.4. CLASSIFICATION DES INCIDENTS / ATTAQUES

Les incidents ou attaques exposés ci-dessus peuvent être classifiés comme suit :

- Les **attaques passives** : le pirate n'effectue qu'une lecture des données sans modification. Ce type d'attaque s'oppose à la confidentialité des données.
- Les **attaques actives** : le pirate cherche à supprimer, ajouter ou transformer les données auxquelles il arrive à accéder. Ce type d'attaque s'oppose à l'intégrité, à l'authenticité des données aussi bien qu'à leur confidentialité.

Une autre classification définit quatre catégories principales d'attaques :

- l'**accès** : ce type d'attaque concerne la confidentialité de l'information.;
- la **modification** : ce type d'attaque est dirigé contre l'intégrité de l'information ;

- le **déni de service** : il s'agit d'attaques qui rendent impossible l'utilisation des ressources par les utilisateurs légitimes ;
- la **répudiation** : il s'agit d'une attaque contre la responsabilité qui consiste à donner de fausses informations ou à nier qu'un événement ou une transaction se sont réellement produit, qu'un message a réellement été reçu ou transmis.

2.3. PROTOCOLES DE SÉCURITÉ

2.3.1. PROBLÈMES ET LIMITES DE LA CRYPTOGRAPHIE

[JAWORSKI 01] • [MAIWALD 01] • [TANENBAUM 97] • [UNIGE 00]

Pour rappel, un *algorithme de chiffrement* est une fonction mathématique utilisée lors du processus de chiffrement et de déchiffrement. Cet algorithme est associé à une *clé* de chiffrement. Le résultat du chiffrement varie selon la clé utilisée.

On peut considérer qu'un chiffrement particulier est *vulnérable* ou *invulnérable*. La vulnérabilité d'un chiffrement se mesure en terme de temps et de ressources nécessaires pour obtenir le texte en clair sans posséder préalablement le mécanisme de déchiffrement. Un chiffrement invulnérable pourrait être défini comme étant particulièrement difficile à déchiffrer sans l'outil de déchiffrement approprié, ou nécessitant des moyens disproportionnés par rapport à l'avantage escompté.

Avec l'évolution permanente des moyens informatiques, l'invulnérabilité d'un chiffrement prend évidemment tout son sens momentané : l'algorithme invulnérable d'hier n'offre plus une sécurité suffisante aujourd'hui. Et que seront demain les algorithmes « invulnérables » d'aujourd'hui ?

Ainsi, la sécurité des données chiffrées repose sur deux piliers fondamentaux : l'**invulnérabilité** de l'algorithme de chiffrement, et la **confidentialité** de la clé utilisée.

Le **chiffrement symétrique** (ou *chiffrement à clé secrète*, dans lequel une seule et même clé suffit à la fois pour le chiffrement et le déchiffrement) offre souvent l'avantage de la simplicité et de la rapidité. Son principal point faible consiste en la garantie de la confidentialité de la clé de chiffrement. En effet, le point critique est le partage de la clé entre l'émetteur du message et son destinataire, qui doivent convenir d'une clé commune en évitant sa divulgation notamment lors de la transmission.

La **distribution des clés** est le problème central de toute la cryptographie à clé secrète : comment faire parvenir sa clé de chiffrement à son destinataire sans qu'elle puisse être interceptée par un tiers ?

C'est de cette difficulté que sont nés les *algorithmes à clé publique*, ou **chiffrement à clé asymétrique**, qui utilisent une paire de clés pour le chiffrement : une donnée chiffrée avec la clé publique ne peut être déchiffrée que par la clé privée correspondante. Ainsi, la clé publique peut être librement diffusée, et servir à chiffrer des données que seul le détenteur de la clé privée pourra lire.

Soulignons également que le procédé peut être inversé : une donnée chiffrée avec la clé privée ne pourra être déchiffrée qu'avec la clé publique correspondante. C'est bien là le but recherché dans le cadre de la **signature** électronique, dans laquelle c'est l'identité du signataire qui doit être attestée. Comme nous l'avons déjà remarqué, cette même technique de chiffrement à clés asymétriques offre donc des garanties de confidentialité, d'authentification, d'intégrité des données et de non-répudiation.

Cette technique présente en outre un avantage majeur : elle ne nécessite pas de dispositif particulier pour l'échange de clés secrètes, ce qui la rend moins coûteuse à mettre en oeuvre. Par contre, l'absence de transmission ou de partage de clés secrètes se paie par une plus grande complexité

d'algorithme et par des calculs beaucoup plus lourds que pour le chiffrement symétrique. Soit un dilemme à l'heure du choix de l'algorithme de chiffrement : rapidité ou résistance de la technique ?

Une solution intermédiaire existe, appelée **système de chiffrement hybride**, conjuguant la facilité et la sécurité du chiffrement à clé publique et la rapidité du chiffrement à clé secrète. L'idée est d'utiliser la technique de chiffrement asymétrique pour la transmission des clés de chiffrement symétrique.

Celui qui envoie des données utilise une clé de session symétrique, valable uniquement pour la session de transmission en cours, pour chiffrer les données à transmettre. La clé de session est elle-même chiffrée avec la clé publique du destinataire, et transmise à ce dernier avec les données chiffrées. Le destinataire utilise alors sa clé privée pour récupérer la clé de session et ainsi déchiffrer les données par un algorithme symétrique conventionnel. Ce système améliore donc globalement la performance et la distribution des clés, sans pour autant compromettre la sécurité.

Le même type de problème de performance se pose dans le cadre de la signature numérique : chiffrer la totalité des données avec la clé privée de l'envoyeur serait très pénalisant en terme de rapidité. On utilise donc un fonction de hachage à sens unique pour créer un **condensé de message** qui sera joint aux données. Seul le condensé de message sera chiffré en tant que signature de l'envoyeur.

Toutefois, quelle que soit la technique de chiffrement utilisée, qu'elle soit symétrique ou asymétrique, la pierre angulaire des algorithmes demeure la clé de chiffrement elle-même. De façon générale, plus la clé est grande, plus la sécurité du chiffrement est élevée. Compte tenu des spécificités des algorithmes, ceci est particulièrement vrai pour le chiffrement asymétrique, qui nécessite des clés particulièrement grandes pour être efficaces en terme d'inviolabilité.

Dans ce contexte, une grande partie de la difficulté du choix d'une technique de chiffrement réside dans la sélection d'une taille de clé appropriée : suffisamment grande pour être sécurisée, mais suffisamment petite pour garantir une certaine rapidité. Et toujours en tenant compte du profil des tiers susceptibles d'essayer de percer le chiffrement, de leur détermination, mais aussi et surtout du temps et des ressources techniques, humaines et financières dont ils disposent...

Sans oublier de s'assurer que le prétendu détenteur de clé est bien l'interlocuteur visé : en effet, si les données chiffrées avec une clé publique ne peuvent effectivement être déchiffrées que par le détenteur de la clé privée correspondant, encore faut-il pouvoir s'assurer que la clé publique utilisée est bien celle de son destinataire présumé, et non celle d'un tiers essayant de se substituer à ce destinataire. C'est la raison d'être des **certificats numériques**.

Un certificat numérique contient des informations liées à une clé publique, permettant d'aider à valider une clé en l'authentifiant. Ces données sont attestées par la signature numérique d'un tiers de confiance appelé **autorité de certification**. Un certificat est donc une clé publique associée à un ou deux identificateurs et estampillée par un ou plusieurs tiers fiables.

En réalité, la difficulté a simplement été déplacée verticalement : comment en effet s'assurer de la validité d'un certificat ? Comment par exemple s'assurer que l'autorité de certification est bien elle-même authentique ? C'est pourtant simple : en y associant également un certificat numérique attesté par une autorité de certification supérieure ! Une hiérarchie d'autorités de certification est donc la seule réponse envisageable à la question de l'authenticité d'un certificat : une autorité « souveraine » délègue le droit de garantir de nouveaux certificats, et tout certificat signé par une autorité est considéré comme valide par les autres certificats de la même hiérarchie.

Un autre aspect de la vérification de la validité consiste à garantir la **durée de validité** d'un certificat. Un certificat est toujours créé avec une période de validité donnée, au terme de laquelle il est réputé expiré et donc non fiable pour les chiffrements postérieurs à sa date de validité.

Un certificat peut également être **révoqué**, c'est-à-dire rendu invalide avant expiration de sa date de validité. Un certificat révoqué est donc particulièrement suspect et non fiable ; il indique en général que les informations d'identification correspondant à la clé publique ne sont plus garanties, ou pire encore que la clé publique a été compromise et n'est donc absolument plus fiable.

Enfin, il faut également souligner la problématique de distribution des certificats et des clés publiques qu'ils authentifient. Comment distribuer publiquement ces informations à toute personne désirant communiquer avec le détenteur de la clé privée correspondante ?

Une réponse à ce problème est la mise en place des mécanismes de stockage centralisés, appelés **serveurs de certificats**, qui permettent de soumettre et de récupérer des certificats. La nécessité de tenir compte également des services complémentaires liés à la gestion des certificats (émission, révocation, fiabilité) conduit à des infrastructures de clé publique ou **PKI** (*Public Key Infrastructure*). Une PKI représente une autorité de certification particulière et prend en charge la gestion des certificats de cette autorité. L'autorité de certification est en réalité l'élément central d'une PKI.

2.3.2. CARACTÉRISTIQUES DES PRINCIPAUX ALGORITHMES

[JAWORSKI 01] ♦ [MAIWALD 01]

Après avoir brossé la problématique des techniques de chiffrement en termes de vulnérabilité et de performance, nous présentons ici les grandes caractéristiques des algorithmes standardisés les plus utilisés pour chacune des techniques abordées. Dans le cadre de notre mémoire, il faut souligner que tous les algorithmes présentés ici (sauf AES) sont supportés en standard par la plateforme Java ; aussi y ferons-nous référence dans les sections y afférents.

FIG. 5 – TABLEAU : PRINCIPAUX ALGORITHMES DE SÉCURITÉ

Chiffrement symétrique	DES (Data Encryption Standard)
Chiffrement asymétrique	RSA (Rivest, Shamir, Adleman)
Condensé de message	MD5 (Message Digest 5) SHA1 (Secure Hash Algorithm)
Certificats	X509
Chiffrement hybride	SSL

2.3.3. CHIFFREMENT À CLÉ SYMÉTRIQUE : DES

Adopté en 1977 comme standard de chiffrement à clé secrète (chiffrement symétrique), DES (*Data Encryption Standard*) a résisté aux attaques jusque dans les années 1990. Depuis lors vulnérable, il reste malgré tout très utilisé, car il offre toujours une sécurité raisonnable dans des applications où le coût engendré par une attaque dépasse la valeur de l'information à protéger.

DES utilise en standard une clé effective de **56 bits** (plus 8 bits de correction d'erreurs, soit 64 bits au total). Le texte est codé par bloc de 64 bits : chaque bloc subit d'abord une permutation indépendante de la clé, suivie d'un chiffrement selon la clé et enfin d'une permutation inverse de la première permutation. DES est conçu pour être utilisé dans plusieurs modes : ECB (*Electronic CodeBook*), CBC (*Cipher Block Chaining*) et sa variante PCBC (*Propagating CBC*), CFB (*Cipher FeedBack*), et enfin OFB (*Output FeedBack*).

Sans entrer dans le détail de chaque mode, soulignons que le choix du mode à utiliser dépend du type de données que l'on désire chiffrer. Une première différence fondamentale est que les quatre premiers

modes (ECB, CBC, PCBC et CFB) sont des modes de chiffrement par bloc (conçus pour chiffrer des blocs de texte), alors que OFB est un mode de chiffrement par flux (conçu pour chiffrer un flux quelconque de bits).

Le mode OFB est adapté lorsque le chiffrement est destiné à protéger des données vocales ou de la vidéo en flux continu. CFB est adapté pour des applications nécessitant le chiffrement de blocs inférieurs à 64 bits (par exemple une application de terminal distant en mode texte). ECB est adapté au chiffrement de blocs de données indépendants (par exemple les champs d'un enregistrement d'une base de données). CBC et PCBC sont utilisables dans tous les autres cas. PCBC a ceci de particulier qu'il propage une erreur de chiffrement à tous les blocs suivants l'erreur, alors que dans CBC l'erreur est circonscrite à 16 octets. Notons que PCBC n'est pas totalement standardisé, ce qui lui fait souvent préférer CBC, complètement normalisé.

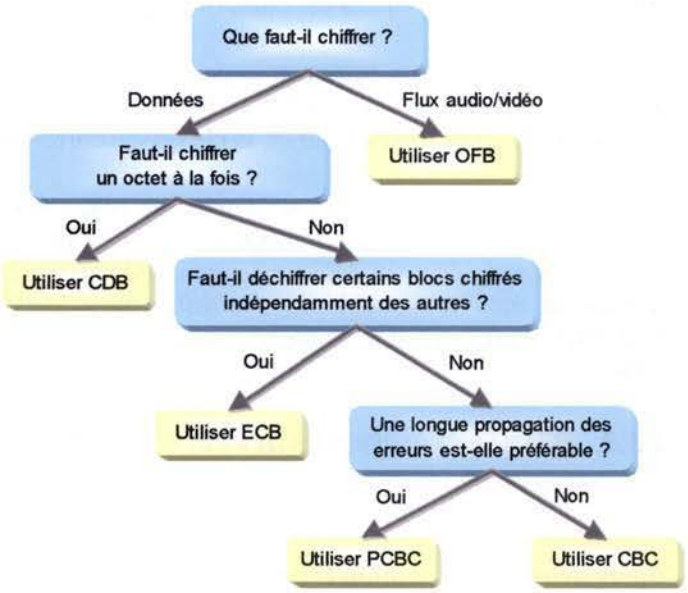


FIG. 6 – ARBRE DE DÉCISION DU DES

Malgré les limitations affichées dès les années 1990, la grande popularité de DES a conduit à rechercher des améliorations de l'algorithme qui préserveraient le chiffrement de base. C'est ainsi qu'est apparu DESede (ou 3DES ou **TripleDES**), une version sophistiquée du DES opérant un chiffrement en trois itérations : le texte en clair est d'abord chiffré par DES en utilisant une première clé A ; le texte chiffré est alors déchiffré en utilisant une seconde clé B, ce qui produit l'équivalent d'un second chiffrement ; puis le texte obtenu est chiffré à nouveau avec une troisième clé C. En utilisant trois clés différentes, la clé efficace de TripleDES atteint alors **168 bits** (une variante de DESede utilise les clés A et C identiques, ce qui donne tout de même une longueur de clé de 112 bits).

AES – ADVANCED ENCRYPTION STANDARD

Ces dernières années, le NIST développe un remplaçant à long terme pour le DES : il s'agit de l'algorithme AES (*Advanced Encryption Standard*), qui s'inscrit comme le nouveau standard pour le chiffrement à clé symétrique, et dont l'usage devrait se généraliser dans les années à venir. C'est l'algorithme Rijndael (du nom de leurs auteurs Vincent Rijmen et Joan Daemen, de l'Université de Louvain) qui a été choisi pour être normalisé sous le nom AES.

Cet algorithme de chiffrement par bloc se caractérise par une longueur de bloc et de clé multiple de 32 bits (Rijndael supporte aussi une longueur de bloc variable, mais cette caractéristique n'a pas été retenue dans le standard). Actuellement le chiffrement par clés de 128, 192 et 256 bits a été spécifié, avec les mêmes longueurs de blocs disponibles (soit neuf combinaisons au total). En fait, AES remplit les mêmes exigences que son prédécesseur DES, tout en étant largement plus sûr et plus flexible.

AES est trop récent pour être distribué en standard dans la plateforme Java ; toutefois le package AES Java peut être téléchargé sur le site officiel de l'AES (http://csrc.nist.gov/encryption/aes/), et installé comme extension des services cryptographiques (selon le principe du CSP; voir *3.1.3 Java Cryptography Architecture*).

2.3.4. CHIFFREMENT À CLÉ PUBLIQUE : RSA

L'algorithme à clé publique le plus utilisé est RSA (d'après les initiales du nom de ses concepteurs : *Rivest, Shamir, Adleman*), introduit dès 1977. RSA est utilisé aussi bien pour le chiffrement des données que pour la construction de signatures électroniques.

RSA est basé sur la difficulté de factorisation des grands nombres. Sans entrer dans les détails mathématiques, le fonctionnement général de l'algorithme est le suivant :

1. Deux nombres premiers (de 100 chiffres ou plus) sont générés : p et q, avec $n = p.q$.
2. Une clé publique entière e est sélectionnée, de manière qu'elle soit première avec (p-1)(q-1).
3. La clé privée d est calculée pour que $e.d \bmod (p-1)(q-1) = 1$.

Le chiffrement est effectué sur des nombres du texte en clair m qui sont inférieurs à n en calculant $m^e \bmod n$. Le déchiffrement s'effectue sur le texte chiffré c en calculant $c^d \bmod n$.

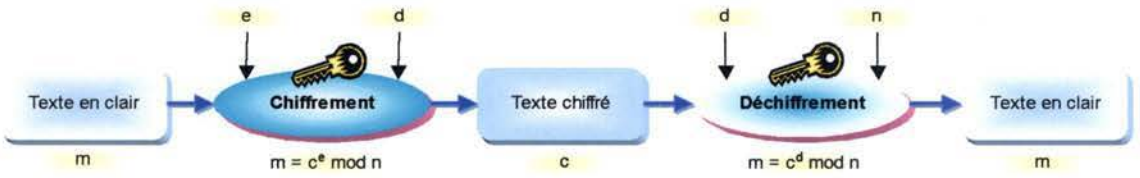


FIG. 7 – CHIFFREMENT AVEC RSA

Les nombre p et q sont les éléments vitaux des clés calculées ; ils doivent être impérativement gardés secrets (ou détruits).

Pour qu'un utilisateur X puisse utiliser RSA, il est nécessaire qu'il diffuse sa clé publique, soit e et n. Un message qui lui serait destiné doit alors être découpé en blocs m_i inférieurs à n, et chiffrés avec $m_{ie} \bmod n$. Notons que les blocs de messages sont généralement des multiples de 64 bits, un remplissage étant utilisé pour les blocs partiels. Lorsque X reçoit les blocs chiffrés c_i, il les déchiffre en calculant pour chaque bloc $c_{id} \bmod n$. Pour autant qu'il ait gardé la valeur de d secrète, X est alors le seul à pouvoir déchiffrer les messages chiffrés avec la clé publique.

Jusqu'ici, RSA a bien résisté aux attaques, même si des clés de plus en plus longues ont pu être cassées jusqu'ici ; chaque bit ajouté à la clé rend celle-ci largement plus difficile à casser. Toutefois une avancée mathématique peut en théorie le rendre obsolète très rapidement, dès lors qu'une méthode efficace de factorisation des grands nombres serait trouvée.

Soulignons que la complexité mathématique évidente par rapport à un algorithme comme DES rend un algorithme tel que RSA beaucoup plus lent à exécuter (de l'ordre de 100 à 1000 fois plus lent). C'est pourquoi RSA sera beaucoup utilisé pour la partie chiffrement asymétrique d'un protocole hybride de chiffrement tel quel SSL (voir *2.3.7. Secure Socket Layer*).

2.3.5. CONDENSÉS DE MESSAGE : MD5 ET SHA-1

Une fonction de condensé de message n'a rien de secret : elle est disponible publiquement, et ne nécessite aucune notion de clé. Ce type de fonction, appelée aussi fonction de hachage, a deux caractéristiques essentielles :

- pour un condensé de message donné, il est pratiquement impossible de calculer un message qui produirait cette valeur de condensé de message ;
- il est pratiquement impossible de trouver deux messages produisant exactement la même valeur de condensé de message.

Les algorithmes de condensés de messages les plus courants sont MD5 (*Message Digest*) et SHA-1 *Secure Hash Algorithm*), tous deux standardisés (MD5 a été adopté par le NIST - *National Institute of Standards and Technology* - en 1993).

Historiquement, MD5 et SHA-1 sont tous deux des évolutions de MD4. SHA-1 a été conçu pour résoudre certains problèmes de vulnérabilité de son prédécesseur.

Si des versions MD1 à MD4 existent, MD5 est de loin la version la plus utilisée. MD5 fonctionne sur des messages de longueur arbitraire et génère un condensé de message de **128 bits** (soit 16 octets).

SHA-1 fonctionne sur des messages dont la longueur peut atteindre 2^{64} bits (ce qui en fait une limite somme toute assez théorique) ; il génère un condensé de message sur **160 bits**.

2.3.6. X.509 – CERTIFICATS NUMÉRIQUES

Le format de certificat ISO X.509 est le plus couramment utilisé, notamment dans des environnements Web. Les certificats X.509 contiennent les informations suivantes :

- La version de X.509 utilisée avec le certificat (actuellement v.3, mais la v.1 est toujours utilisée) ;
- Le nom de l'entité et sa clé publique ;
- Une plage de dates de validité du certificat ;
- Un numéro de série attribué par l'autorité de certification ;
- Le nom de l'autorité de certification ;
- Une signature numérique créée par l'autorité de certification.

2.3.7. SSL - SECURE SOCKET LAYER (SSL / TLS)

[GARMS 01] ♦ [NETSCAPE 98] ♦ [CRYPTOSEC 00]

SSL a été développé à l'origine par Netscape, comme protocole de communication sécurisé sur les réseaux de type TCP/IP ; il s'est très vite répandu pour des applications HTTP sur le Web. La première version diffusée est SSL v2.0 (en 1994), améliorée en SSL v3.0 (en 1996). Cette version a servi de base au développement du protocole **TLS** (*Transport Layer Security*), destiné à être normalisé par l'IETF (*Internet Engineering Task Force*). TLS correspond à la version SSL v3.1.

D'un point de vue architecture réseau, SSL s'insère donc entre la couche Réseau (protocole TCP/IP) et la couche Application (les protocoles de plus haut niveau comme HTTP, LDAP, IMAP).

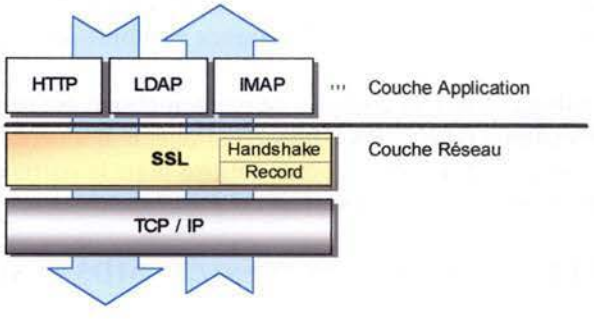


FIG. 8 – SSL DANS L'ARCHITECTURE RÉSEAU

SSL est un protocole à négociation, basé sur l'authentification d'un client (typiquement, un navigateur web) et d'un serveur, avec chiffrement de la session établie entre eux.

Ainsi, SSL ne dépend pas des applications utilisées, et s'applique en dessous de protocoles tels que HTTP ou FTP. Le protocole HTTP sécurisé par SSL est appelé HTTPS, natif dans la plupart des browsers.

SSL supporte essentiellement les algorithmes d'échange de clés basés sur RSA et Diffie-Hellman, des certificats numériques à la norme X.509, ainsi que des algorithmes de chiffrement tels que DES, Triple-DES, RC4, et les condensés SHA-1 et MD5. Notons encore que SSL utilise des codes MAC (*Message Authentification Code*) pour contrôler l'intégrité des données (ou plus exactement HMAC, dans lesquels la clé secrète est ajoutée aux données avant hachage).

SSL peut lui-même être décomposé en deux couches principales :

- un *Handshake Protocol*, qui gère l'établissement de la connexion elle-même (authentification, négociation des clés, des protocoles de chiffrement et de condensé de message), avant l'échange de données proprement dit ;
- un *Record Protocol*, qui joue le rôle d'interface avec les couches inférieures du modèle réseau ; il garantit la confidentialité et l'intégrité des données transmises (après négociation d'une clé de chiffrement partagée, par le protocole de négociation).

On distinguera encore un protocole *Alert*, qui spécifie les messages d'erreur qui peuvent survenir pendant la communication entre le client en le serveur. Ces messages sont composés deux octets, dont le premier indique s'il s'agit d'une erreur de type « warning » ou « fatal », le second octet représentant un code d'erreur. S'il s'agit d'une erreur fatale, la connexion est immédiatement abandonnée.

Les erreurs fatales concernent la réception d'un message non reconnu par le protocole, d'un paramètre incorrect dans un des messages, de la détection d'un code MAC incorrect ou encore d'une erreur lors du déchiffrement. Les erreurs de type « warning » concernent essentiellement des problèmes liés aux certificats (signature du certificat non valide, certificat révoqué ou expiré, etc.).

HANDSHAKE PROTOCOL.

L'objectif du protocole d'établissement de connexion est de créer une clé de session symétrique, partagée entre les parties, après que le serveur se soit authentifié auprès du client (et éventuellement que le client se soit authentifié auprès du serveur).

Pour cela, SSL utilise les techniques des clés asymétriques, lesquelles offrent les services d'authentification et de confidentialité. La clé symétrique construite pour la suite de la communication offre l'avantage d'une plus grande rapidité de chiffrement/déchiffrement.

Ce protocole de négociation s'exécute par l'échange d'une suite particulière de messages entre le client et le serveur, dont certains sont optionnels, et qui peuvent être regroupés en quatre phases distinctes.

Nous décrivons ci-après les quatre phases typiques d'une négociation SSL.

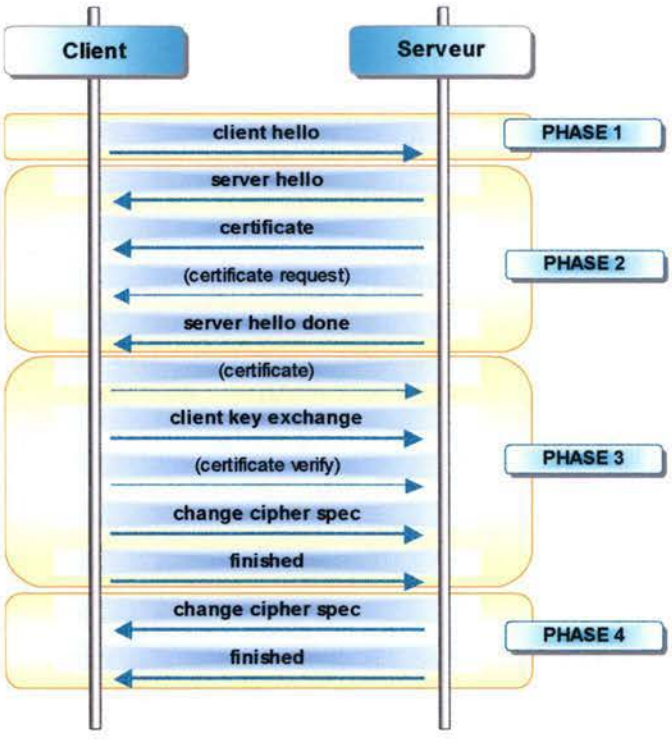


FIG. 9 – SSL : HANDSHAKE PROTOCOL

Phase 1 : ETABLISSEMENT DES PARAMÈTRES DE SESSION.

L'échange des messages débute avec une requête « hello » en provenance du client ; cette requête a pour objectif de renseigner le serveur sur les paramètres supportés par le client, c'est-à-dire :

- le numéro de la plus haute version de SSL reconnue par le client ;
- la liste des algorithmes d'échange de clés et de chiffrement supportés par le client ;
- la liste de algorithmes de compression supportés par le client ;
- un ID de session ;
- une série de 32 octets de données générés aléatoirement par le client pour cette seule session (destinés à prévenir les rejeux de paquets).

Phase 2 : AUTHENTIFICATION DU SERVEUR ET ÉCHANGE DES CLÉS.

Le serveur répond à son tour par une requête de type « hello », indiquant les choix parmi les paramètres proposés par le client : version SSL, algorithmes d'échange de clés et de chiffrement, algorithme de compression, ainsi que l'ID de session et 32 bytes générés aléatoirement par le serveur.

Le serveur envoie alors un certificat (à la norme X.509, signé par une autorité de certification). Ce certificat contient notamment la clé publique du serveur, ce qui permet au client d'envoyer des messages chiffrés que seul le serveur, détenteur de la clé privée correspondante, pourra déchiffrer.

Le serveur peut également demander l'authentification du client, en lui réclamant également un certificat. Toutefois cette demande est optionnelle, et n'est normalement pas requise pour une connexion de type HTTPS. Le client est alors dit *anonyme*, tandis que le serveur est authentifié.

Enfin, le serveur envoie un message « hello done » pour signifier qu'il a terminé sa communication et qu'il attend une réponse du client.

Phase 3 : AUTHENTIFICATION DU CLIENT ET ÉCHANGE DES CLÉS.

Le client vérifie les informations reçues du serveur, et notamment la validité du certificat ; ce qui constitue évidemment un point délicat du protocole. Si le serveur demande un certificat client, ce dernier envoie à ce moment son certificat X.509 ; si le client ne dispose pas d'un certificat, il envoie un message « no certificate ».

Le client envoie alors un message « client key exchange », qui constitue (dans le cas de RSA, de loin le plus utilisé) en une suite de 48 bytes générés aléatoirement qui servira à la construction d'une clé de session commune au client et au serveur. Cette suite de bytes est générée au départ des données aléatoires déjà échangées auparavant. Ces 48 bytes sont chiffrés (typiquement par un algorithme RSA) avec la clé publique du serveur qui a été communiquée au client au travers du certificat serveur.

Dans le cas où un certificat client était requis par le serveur, un message « certificate verify » est envoyé, qui consiste en un condensé de message calculé sur base de l'ensemble des messages déjà échangés avec le serveur, chiffré avec la clé privée du client. Ceci permettra au serveur de vérifier que le client est bien le détenteur du certificat client envoyé.

Le client calcule alors une clé de chiffrement (typiquement avec RC4), et un code MAC pour le contrôle d'intégrité des données. Les clés sont générées à partir du nombre aléatoire déjà échangé entre le client et le serveur. Le client envoie alors un message « change cipher spec », qui indique au serveur que les prochains messages utiliseront ces clés de chiffrement et de contrôle d'intégrité. Il est à noter que le client ne communique pas la clé de session au serveur, celui-ci étant capable de générer la même clé de son côté grâce au nombre aléatoire échangé et aux paramètres fixés.

Enfin, le client envoie un message de terminaison, chiffré avec la clé de session et comprenant un code MAC pour le contrôle d'intégrité.

Phase 4 : FIN DE LA NÉGOCIATION.

Le serveur répond au client avec son propre message « change cipher spec » (en fait, un octet), qui signifie à son tour au client que la suite de la communication sera chiffrée avec la clé de session. Le protocole de négociation prend fin avec l'envoi du message de terminaison du serveur (« finished »), chiffré avec la clé de session et comprenant un code MAC.

A partir de ce point, toute la suite de la communication sera effectuée avec une méthode chiffrement symétrique basée sur la clé de session calculée par les deux parties.

2.4. LANGAGES ORIENTÉS OBJETS ET SÉCURITÉ

[BLAKLEY 99]

Nous voudrions mettre en évidence certains problèmes spécifiques liés à la sécurité des langages orientés-objet. Après avoir rappelé brièvement les constituants de ces langages, nous montrerons les défis à relever pour que la sécurité soit **assurée** et **aisément gérable** par l'administrateur du système. Par la suite, lorsque nous aurons décrit les techniques de sécurité du langage Java, nous pourrons analyser dans quelle mesure Java répond aux exigence qui découlent de ces problèmes spécifiques.

2.4.1. CONSTITUANTS DES LANGAGES ORIENTÉS-OBJET

Un langage orienté-objet utilise trois ingrédients de base : les classes, les objets et les messages :

- Les **classes** sont comparables à une recette : elles décrivent comment construire un objet et comment réaliser ce qu'il est capable de faire.
- Un **objet** est créé au départ de la description contenue dans une classe (= **instanciation**). Chaque objet ainsi créé est appelé **instance** de la classe. Il est possible de créer plusieurs instances d'une même classe ; on obtient alors plusieurs objets du même type, mais il s'agit d'objets différents.

  Les objets contiennent à la fois des procédures et des données ; les procédures, appelées **méthodes**, c'est ce que l'objet est capable de faire ; les données, appelées **attributs**, c'est ce que l'objet est capable de mémoriser.
- Un objet peut envoyer des **messages** à un autre objet pour exécuter une des ses méthodes ou obtenir la valeur d'un de ses attributs. Cette communication par message cache en fait une réalité complexe : les objets peuvent se trouver sur des machines différentes, ne pas être actifs, ne pas parler le même langage, etc. Nous l'étudierons plus en détails RENVOI-> applications distribuées.

2.4.2. SÉCURITÉ DES OBJETS ET PROBLÈMES PARTICULIERS

La sécurité des systèmes orientés-objet est similaire à celle des autres systèmes. On y retrouve en effet les mêmes besoins de confidentialité, d'authentification, de permissions d'accès, de disponibilité du service et d'audit.

La sécurité est assurée par une série de règles et de permissions qui permettent au système de déterminer si un principal peut accéder à une ressource.

Néanmoins, la sécurité des systèmes orientés-objets introduit certaines difficultés supplémentaires :

- **Nommage (*Naming*)**

  Les systèmes orientés-objet n'ont pas de système d'attribution de noms hiérarchique et rigide. En effet, il existe beaucoup d'objets avec alias, d'objets anonymes, ou encore d'objets avec plusieurs noms. Lorsqu'un objet est instancié, le système réserve un bloc mémoire correspondant à la description figurant dans la classe. Le système crée ensuite une référence qui sera utilisée pour retrouver le bloc mémoire correspondant au nouvel objet
  Cette référence (généralement une adresse mémoire, de la forme `0x1034AF01`) est unique par rapport à la machine qui l'a créée, mais sans garantie qu'une autre machine ne puisse utiliser la même référence.

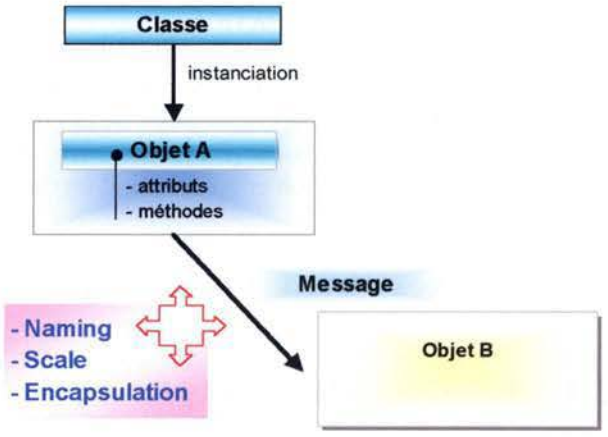


FIG. 10 – SÉCURITÉ ET POO

  La plupart des systèmes orientés-objet fournissent un *naming service*, permettant d'attacher un nom à un objet. Ce type de service interdit l'utilisation du même nom pour deux objets différents, mais il permet d'utiliser plusieurs noms pour le même objet.

  Cette difficulté d'associer un nom unique à un objet rend plus difficile l'écriture des permissions.

  Exigence : Les systèmes orientés-objet sécurisés devraient permettre à l'administrateur de définir des règles de sécurité sur un objet sans connaître sa référence. Les objets anonymes et les objets possédant plusieurs noms devraient être sécurisables par la même règle de sécurité sans devoir se soucier de leur référence.

- **Échelle**

  Le nombre d'objets et de types d'objets différents est beaucoup plus élevé que dans les systèmes procéduraux. Un projet orienté-objet type peut contenir plusieurs dizaines de milliers d'objets, parfois même plusieurs millions. Cela rend impossible la tâche de l'administrateur si l'on veut gérer l'accès à chaque objet. De plus la vérification d'un nombre de règles aussi élevé alourdirait considérablement les temps de réponse de l'application. Il est donc indispensable de gérer l'accès aux ressources par groupes d'objets ayant des besoins similaires en sécurité.

  Les systèmes procéduraux définissent un nombre limité de type de ressources (fichiers, bases de données) et d'actions associées à ces ressources (lecture, écriture, création, suppression). En général, les types de ressources et les actions associées sont prédéfinies et ne peuvent pas être étendues.

  A l'opposé, les systèmes orientés-objet laissent au programmeur la possibilité de créer un grand nombre d'objets de type différents en créant de nouvelles classes. Chaque classe fournit des méthodes personnalisées et spécifiques à la classe.

  D'un point de vue sécurité, cela signifie qu'il est possible de créer des règles spécifiques et adaptées à chaque ressource. Mais cela signifie aussi que l'administrateur doit comprendre ce que

fait chaque méthode pour être capable de déterminer qui peut être autorisé à exécuter cette méthode.

Exigence : Les systèmes orientés-objet sécurisés devraient permettre à l'administrateur de contrôler l'accès à une méthode particulière d'un objet. Néanmoins, il ne devrait pas être obligé de comprendre ce que fait chaque méthode pour définir les règles de sécurité. Dès lors, le système devrait permettre à l'administrateur d'appliquer une seule règle de sécurité pour un ensemble de méthodes similaires.

- **Encapsulation**

Les objets contiennent procédures et données, mais dans une approche orientée-objet correcte, les données ne sont pas accessibles directement. Cette propriété des objets s'appelle l'encapsulation : les données contenues dans un objet sont cachées et ne sont accessibles que par les méthodes de l'objet. L'encapsulation permet de modifier la représentation interne des données sans remettre en cause le reste du programme.

Ce concept est un des piliers de la programmation orientée-objet, mais, du point de vue de l'administrateur, il cache des informations importantes sur la structure du système. Sans ces informations, il est difficile d'imaginer quelles sont les permissions nécessaires pour protéger le système.

Par ailleurs, nous venons de voir que les objets ne sont pas manipulés directement mais par l'intermédiaire d'une référence. Ce qu'il est important de protéger, ce n'est pas seulement la référence mais surtout le bloc mémoire où est stocké l'objet. Imaginons une carte au trésor : mettre cette carte au trésor dans un coffre n'empêche pas d'accéder au trésor lui-même.

Ajoutons que durant l'exécution d'une application, le flux de bytes correspondant physiquement à un objet peut aussi être stocké sur disque ou transiter sur un réseau, ce qui le rend plus accessible et donc plus vulnérable.

Exigence : Les systèmes orientés-objet sécurisés devraient permettre à l'administrateur de définir des règles de sécurité sans connaître les détails d'implémentation des attributs et des méthodes d'un objet. Lorsque le flux physique de bytes correspondant à un objet devient accessible, le système devrait fournir des moyens de protection adéquats.

3. SÉCURITÉ DANS JAVA

3.1. ARCHITECTURE DE SÉCURITÉ

3.1.1. LA SÉCURITÉ ET JAVA

[BONJOUR 00] • [JAWORSKI 01]

Java se distingue des autres langages de programmation par le fait qu'il a été, dès le départ, conçu pour être utilisé dans un environnement distribué avec utilisation de code et d'objets distants. La sécurité revêt par conséquent un intérêt particulier évident dans un environnement dont une des principales caractéristiques est de pouvoir télécharger du code distant pour l'exécuter localement de façon transparente, alors même que l'on ne sait rien ni de l'origine du code ni des intentions de ses auteurs. Peut-on vraiment prendre le risque d'exposer toutes les ressources locales à n'importe quelle applet ?

Dans une telle perspective, les aspects de sécurité ont donc été traités *a priori* lors de la conception du langage, et non *a posteriori* comme dans d'autres langages. Il n'est donc pas étonnant de retrouver ces préoccupations à tous les niveaux de conception de la plateforme Java. Le modèle de sécurité de Java n'a cessé d'évoluer au fil des versions du JDK.

Dans le modèle JDK 1.0, Java introduisait le concept de *sandbox* (« bac à sable »), modèle très restrictif qui laissait l'accès complet aux ressources pour le code local, mais n'autorisait qu'un accès très limité aux ressources pour du code provenant d'une source distante. Une applet ne pouvait par exemple pas accéder au système de fichiers, ni ouvrir une connexion réseau.

Dans le modèle suivant, le JDK 1.1, ce système du « tout ou rien » fut remplacé par un *modèle d'approbation*, qui introduit le concept fondamental de *signature de code* et d'applet. L'idée est de pouvoir indiquer qu'un code signé par certains fournisseurs est autorisé à accéder aux ressources qu'il demande, tout code provenant d'une source non explicitement autorisée étant confiné à la sandbox.

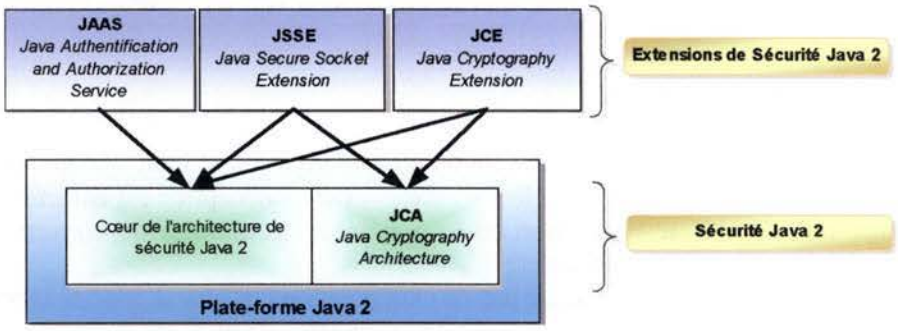


FIG. 11 – PLATEFORME JAVA 2

La plateforme Java 2 (JDK 1.2) introduit un modèle de sécurité affiné et beaucoup plus souple, en donnant la possibilité de définir des domaines de ressources sur lesquels peuvent être établies des

stratégies de sécurité configurables. Java 2 généralise enfin le problème de sécurité en gommant la distinction entre code local et code distant, permettant la construction de solutions globales de sécurité.

L'architecture de sécurité de Java 2 est illustrée dans le schéma ci-dessus (d'après [JAWORSKI 01]) ; la plateforme standard JDK 1.2 intègre le cœur de l'architecture de sécurité de Java 2, auquel est adjoint le composant JCA (*Java Cryptography Architecture*). Les extensions standard de cette plateforme sont JAAS (*Java Authentication and Authorization Service*), JSSE (*Java Secure Socket Extension*) et JCE (*Java Cryptography Extension*). Ces extensions étaient notamment distinguées de la plateforme standard en raison de dispositions légales à l'exportation alors en vigueur aux Etats-Unis. Notons que tous ces composants sont intégrés en standard dans la plateforme JDK 1.4 (version officielle de février 2002).

3.1.2. CŒUR DE L'ARCHITECTURE DE SÉCURITÉ JAVA 2

[BONJOUR 00] • [HORSTMANN 00] • [JAWORSKI 01]

On distinguera trois grands mécanismes de sécurité dans Java :
- La conception du langage et de la JVM (vérificateur de bytecodes et chargeur de classe) ;
- Le contrôle d'accès aux ressources (notamment la protection des fichiers et des accès réseaux) ;
- La signature du code.

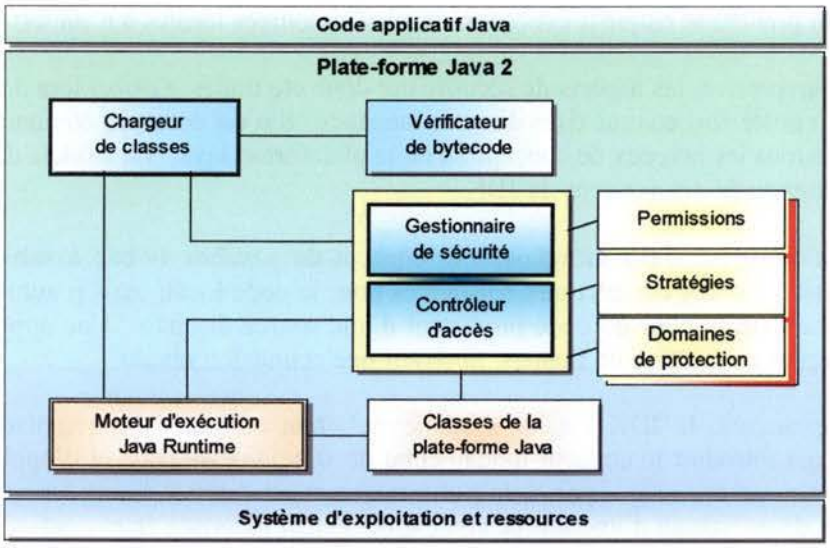


FIG. 12 – ARCHITECTURE DE SÉCURITÉ JAVA 2

LA SÉCURITÉ AU CŒUR DE JAVA

Les spécifications strictes du langage Java sont un facteur augmentant le niveau global de sécurité, par comparaison à d'autres langages (comme le C++ p.ex.). Cette conception est caractérisée par :
- Un typage fort de chaque instruction, avec possibilités restreintes de conversions de types et impossibilité de conversions implicites ;
- L'absence de notion de pointeur (donc pas de manipulation possible d'adresse mémoire) ;
- La gestion automatique de la mémoire, avec un mécanisme de *garbage collector* ;
- Contrôle d'accès aux variables et méthodes, avec 4 niveaux de protection (public, (default), protected, private) et un mécanisme particulier d'interdiction de redéfinition (final) ;
- Pas de surcharge des opérateurs, ni d'héritage multiple.

Dès la compilation (i.e. la génération de bytecode pour la machine virtuelle Java), puis ensuite à travers le **vérificateur de bytecode**, la conformité du code aux spécifications du langage est contrôlée, avec notamment d'autres types de tests tels que l'utilisation des méthodes avec un nombre d'arguments et des types corrects, le contrôle de débordement de pile, l'initialisation obligatoire de toute variable, ou encore la vérification du dépassement des bornes dans un tableau.

Notons que les contrôles opérés par le vérificateur de bytecode sont dépendants de la machine virtuelle. En effet, la JVM définit précisément les types de données primitifs, les structures de vecteurs de types de base, la structure de la pile, ainsi que le jeu d'instructions de la machine : empilement des constantes, manipulation des registres, instructions ALU, transfert de contrôle, invocation de méthode, accès aux variables d'instance, opérations de conversion de type (casting), etc.

Le vérificateur de bytecode prépare également l'exécution de la classe chargée, en partitionnant l'espace mémoire de la machine : mémoire objets, pool des constantes, description des bytecodes, zone d'activation des méthodes (avec pile et registres locaux).

Le vérificateur de bytecode est chargé de tester le format du fichier de bytecode qui constitue la classe à charger ; ce fichier de classe doit en effet être conforme à une structure particulière. Le fichier doit être complet, sans ajout ni élimination, avec une zone de constante conforme aux spécifications. La cohérence même des structures et des références est vérifiée : pas d'héritage illégal, pas de classe sans super-classe, pas de noms de méthodes ou d'attributs mal formés ou de signatures illégales.

Le modèle d'exécution de chaque méthode est également vérifié : l'état de la pile est un invariant en un point donné du programme (ce qui nécessite la construction d'un modèle d'exécution de la classe pour simuler les cas possibles !), les registres locaux ne sont pas utilisés illicitement, les méthodes sont invoquées avec des arguments corrects, les variables d'instance sont manipulées conformément à leur type, les bytecodes eux-mêmes sont cohérents par rapport aux registres, à la pile et aux constantes. Nous n'entrerons pas davantage dans le détail de cette étape de vérification (le lecteur intéressé trouvera une description plus étendue dans [BONJOUR 00], chapitre 27).

Enfin, après avoir encore remplacé certaines instructions par une version optimisée, le vérificateur de bytecodes donne son aval quand à la validité de la classe chargée. Le flux de données est alors rendu au chargeur de classe.

Soulignons que le **chargeur de classe** est celui qui avait initialement appelé le vérificateur de bytecode. En effet, la première tâche du chargeur de classe (après la localisation du fichier lui-même) est de charger aveuglément un fichier de classe sous la forme d'un vecteur de bytes et de transférer ce vecteur au vérificateur de bytecode. Si ce dernier avalise le code, le chargeur de classe peut alors opérer la *résolution* de la classe, c'est-à-dire le chargement des classes dépendantes de la classe courante (super-classe et classes utilisées par les variables d'instance).

Le chargement de classe et le mécanisme particulier de la délégation lors du chargement d'une classe fait l'objet d'une description détaillée dans une section ultérieure (voir *3.2. Zoom 1 : Chargeur de Classe*).

LE CONTRÔLE D'ACCÈS AUX RESSOURCES

Afin de contrôler l'accès aux ressources et les autorisations qui y sont liées, Java intègre un **Gestionnaire de Sécurité**. Il n'y a qu'une seule instance du gestionnaire de sécurité exécutée dans une instance de machine virtuelle donnée, et son remplacement peut être interdit (selon la configuration de la JVM) pour toute la durée de vie de la JVM. C'est ce qui se passe lors du chargement du gestionnaire de sécurité dans la plupart des browsers web : chargé avant la première applet Java, aucun autre ne peut s'y substituer (pour éviter d'être remplacé par un code malveillant).

Java 2 offre un gestionnaire de sécurité par défaut, configurable selon les besoins. Son utilisation peut être requise via la ligne de commande au démarrage de l'application Java. Les méthodes d'un *Security Manager* permettent de vérifier si l'accès est autorisé sur une ressource particulière. Toutefois, depuis Java 2, les contrôles d'accès réels sont situés dans un *Access Controller*. Il n'est plus conseillé de spécialiser un objet Security Manager personnalisé (ses méthodes sont par ailleurs déclarées *protected*). A la place, Java fournit une hiérarchie de fonctions de permissions extensibles.

Le concept de **permission** s'applique à une ressource sur laquelle un accès particulier peut être accordé ou refusé, ou sur laquelle certaines opérations peuvent ou ne peuvent pas être effectuées. Les permissions sont classées par type, selon une hiérarchie dérivée des classes de base. Ainsi, toutes les classes de permissions dérivent d'une classe abstraite *Permission*, qui forme un objet reprenant le nom d'une ressource et un ensemble de noms d'actions exécutables sur la ressource (p.ex. « read,write »).

Les types de permissions concernent par exemple l'accès aux fichiers, l'accès aux connexions et aux ports réseaux, l'accès à certaines ressources du système d'exploitation (comme le presse-papiers), la possibilité de gérer des threads, la possibilité d'intervenir sur la gestion de la machine virtuelle (en l'arrêtant, par exemple), ou encore la permission de... gérer des permissions !

Java 2 permet de définir des types de permissions personnalisées, directement en dérivant la classe abstraite *Permission*, ou au départ de sa sous-classe *BasicPermission* qui possède déjà une implémentation par défaut de certaines méthodes, pour les permissions de type tout ou rien, sans liste d'actions.

Les permissions qu'elles soient pré-définies ou personnalisées, peuvent être accordées à un code particulier en fonction de son origine (URL) et de sa signature éventuelle. Dans l'implémentation par défaut de la gestion de la sécurité dans Java 2, une stratégie de sécurité, c'est-à-dire l'ensemble des permissions accordées aux différentes sources de code, se définit dans un fichier texte (fichier *java.policy*). L'environnement de Java 2 est fourni avec `policytool`, un outil en mode graphique qui permet l'édition de ce fichier.

Le contrôle d'accès, la gestion des permissions et des fichiers de stratégies de sécurité seront décrites plus en détail dans une section ultérieure (voir *3.3. Zoom 2 : Gestionnaire de Sécurité et Contrôleur d'Accès*).

LA SIGNATURE DE CODE

L'utilisation de technologies d'authentification trouve une de ses applications majeures dans la signature de programmes à exécuter. Cette technique permet de garantir la provenance et l'intégrité d'un programme chargé depuis une source distante.

La signature du code peut être associée aux techniques de gestion des permissions, permettant ainsi de maîtriser la gestion des accès accordés à un programme selon une origine elle-même contrôlée. Ceci permet par exemple de distinguer efficacement des applications signées provenant d'une source intranet ou au contraire internet, en accordant des permissions spécifiques selon l'une ou l'autre.

Dans la section suivante, nous parcourons rapidement l'éventail des technologies regroupées dans le package JCA, y compris les techniques d'authentification proposées nativement par Java.

3.1.3. JAVA CRYPTOGRAPHY ARCHITECTURE

[GARMS 01] ◆ [JAWORSKI 01]

JCA fut proposé avec le JDK 1.1 pour offrir des fonctions de cryptographie de base, à des fins de protection des données, d'identification des intervenants, de gestion de clés et de certificats, ou encore pour la mise en œuvre d'un ensemble de classes d'algorithmes cryptographiques redéfinissables et extensibles. En réalité, JCA n'est pas utilisé pour des opérations de chiffrement ou de déchiffrement, toutes ces fonctions spécifiques ayant été regroupées dans l'extension JCE (voir *3.1.6. Java Cryptography Extension*).

L'architecture JCA propose une série de classes regroupant des fonctions liées à :
- des interfaces ouvertes de fournisseur de service JCA (*java.security*)
- la gestion des certificats (*java.security.cert*),
- la gestion des clés publiques et privées DSA et RSA (*java.security.interfaces*),
- des interfaces de description des algorithmes de gestion des clés (*java.security.spec*).

L'un des intérêts majeurs de JCA réside dans le concept de **fournisseur de service cryptographique** ou CSP (*Cryptography Service Provider*), qui permet à tout fournisseur de service d'étendre JCA au moyen d'implémentations des fonctions cryptographiques en conformité avec les interfaces définies dans JCA. Un CSP étend la classe abstraite *Provider*, qui encapsule ses informations spécifiques.

Un CSP implémente donc un ou plusieurs moteurs cryptographiques, constitués d'algorithmes particuliers et de leurs paramètres spécifiques, selon une interface standard fournie par JCA. Chaque interface d'un fournisseur de service (SPI, *Service Provider Interface*) est étendue par une API de moteur cryptographique utilisable par les programmeurs. Chaque API implémente une méthode particulière *getInstance()* qui prend en paramètre le nom d'un algorithme proposé par cette classe et renvoie une instance du moteur demandé avec l'algorithme requis. Et pour chaque API on peut spécifier le nom d'un CSP particulier à utiliser.

Les types de moteurs cryptographiques fournis en standard par JCA sont résumés dans le tableau ci-dessous, avec les moteurs et algorithmes fournis par le CSP par défaut livré avec Java 2 (« SUN »).

FIG. 13 – TABLEAU : TYPES DE MOTEURS CRYPTOGRAPHIQUES DE JCA

Moteur	Fonctionnalité	SUN
MessageDigest	Création et vérification de digests de message	**MD5, SHA1**
Signature	Création et vérification de signatures numériques	**DSA**
KeyPairGenerator	Génération de paires de clés publiques et privées	**DSA**
KeyFactory	Conversion des spécifications en clés publiques et privées	**DSA**
KeyStore	Gestion d'un référentiel sécurisé de stockage des clés	**JKS**
CertificateFactory	Génération des certificats et des listes de révocation	**X.509**
AlgorithmParameters	Codage des paramètres des algorithmes de chiffrement	**DSA**
AlgorithmParameterGenerator	Création des paramètres d'algorithme de chiffrement	**DSA**
SecureRandom	Création de nombres aléatoires	**SHA1PRNG**
Notes :	▪ JKS – *Java Key Store*, algorithme propriétaire de SUN ▪ SHA1PRNG est standardisé par l'IEEE	

Les fournisseurs installés sont ajoutés dans un fichier texte *java.security*, qui reprend les noms pleinement qualifiés des CSP qui étendent la classe Provider. Ce fichier sert à définir un ordre de préférence pour un algorithme éventuel pour lequel plusieurs implémentations seraient disponibles avec le cumul des différents CSP. Il est également possible de gérer ce fichier par programme.

3.1.4. JAVA AUTHENTICATION AND AUTHORIZATION SERVICE

[GARMS 01] • [JAWORSKI 01]

Java Authentication and Authorization Service permet d'étendre le modèle de sécurité de Java 2 en y ajoutant des mécanismes d'authentification et de gestion des autorisations. L'authentification consiste à vérifier une identité ; l'autorisation consiste à limiter l'accès aux ressources sur base de cette identité. JAAS offre une interface stable et standard, quels que soient les modèles d'authentification et d'autorisation proposés par les fournisseurs de services.

Le modèle de sécurité de base de Java dispose des techniques de signature de code pour déterminer l'origine et le *signataire* du code ; JAAS y ajoute la possibilité de contrôler *l'exécuteur* du code. Cette interface rend en outre le module d'authentification presque indépendant du reste de l'application.

JAAS implémente une version Java du module **PAM** (*Pluggable Authentication Module*), qui offre des mécanismes simples tels que l'authentification par nom d'utilisateur / mot de passe, mais aussi des protocoles plus complexes tels que Kerberos ou d'autres implémentations PKI comme RSA.

L'architecture JAAS propose une série de classes regroupant des fonctions liées à :
- des services de base pour l'authentification et l'autorisation (*javax.security.auth*) ;
- la gestion des données liées au module de login de l'application (*javax.security.auth.callback*) ;
- la connexion à un domaine de sécurité (*javax.security.auth.login*) ;
- une interface de connexion implémentée par les fournisseurs de services (*javax.security.auth.spi*).

Le tableau suivant propose les principales classes et interfaces de ces paquetages.

FIG. 14 – TABLEAU : PRINCIPALES CLASSES ET INTERFACES DE JAAS

Classe	Fonctionnalité
Subject	Entité « utilisateur », avec plusieurs identités et références privées / publiques
LoginContext	Interface de base pour la gestion de la connexion au système
LoginModule	Interface que les fournisseurs de services doivent implémenter
Configuration	Entité de configuration d'une application avec des modules de connexion
CallbackHandler	Interface que les applications doivent implémenter pour l'authentification
Callback	Regroupe les informations utiles pour une implémentation de CallbackHandler
Policy	Stratégie système pour la gestion d'autorisation basée sur un sujet authentifié
AuthPermision	Permissions utilisées durant l'identification

3.1.5. JAVA SECURE SOCKET EXTENSION

[JAWORSKI 01]

Java Secure Socket Extension propose une interface standard aux protocoles sécurisés de type SSL. JSSE gère les versions SSL v3.0 et TLS v1.0. JSSE fournit également une interface de fournisseur de service permettant différentes implémentations des protocoles.

Les classes principales JSSE sont groupées dans *javax.net.ssl*. JSSE utilise les classes de *javax.net* pour la gestion des sockets client et serveur, et les classes de *javax.security.cert* pour la gestion de base des certificats. SSL a fait l'objet d'une description détaillée dans la partie consacrée aux protocoles de sécurité. Son implémentation Java est décrite dans une section ultérieure (voir *3.4. Zoom 3 : Secure Socket Layer*).

3.1.6. JAVA CRYPTOGRAPHY EXTENSION

[GARMS 01] ◆ [JAWORSKI 01]

Java Cryptography Extension regroupe les fonctionnalités purement cryptographiques de la plateforme Java 2. Dans la lignée de la philosophie générale de la sécurité Java, la principale caractéristique de cette extension est le support de fournisseurs de services cryptographiques, permettant de développer des applications indépendantes de l'implémentation des algorithmes cryptographiques particuliers. De plus, le principe de CSP autorise que les algorithmes de chiffrement utilisés dans les applications ne soient pas limités à ceux proposés par Sun.

L'extension JCE propose des classes et interfaces groupant des fonctions liées à :
- l'implémentation des algorithmes cryptographiques de base (14 classes dans *javax.crypto*) ;
- la gestion particulière des clés Diffie-Hellman (3 interfaces dans *javax.crypto.interfaces*) ;
- la spécification de clés et de paramètres d'algorithmes (*javax.crypto.spec*).

Le tableau suivant propose les principales classes cryptographiques de *javax.crypto*, le cœur de JCE.

FIG. 15 – TABLEAU : PRINCIPALES CLASSES CRYPTOGRAPHIQUES DE *javax.crypto* (JCE)

Classe	Fonctionnalité
Cipher	Encapsule les opérations de chiffrement / déchiffrement proprement dit
CipherInputStream	Déchiffrement *Cipher* d'un flux de données en entrée *InputStream*
CipherOutputStream	Chiffrement *Cipher* d'un flux de données en sortie *OutputStream*
SealedObject	Gestion des objets sérialisés et chiffrés (voir *3.5. Zoom 4 : Signing, Sealing & Guarding Objects*)
SecretKey	Extension de *java.security.Key* pour la cryptographie à clé secrète symétrique
SecretKeyFactory	Fabrique de clés secrètes selon leurs spécifications
KeyGenerator	Génération de clés pour le chiffrement symétrique à clé secrète
KeyAgreement	Encapsule un protocole d'échange de clés
Mac	Encapsule un algorithme de code d'authentification de message (MAC)
Notes :	Les classes *Cipher*, *SecretKeyFactory*, *KeyGenerator*, *KeyAgreement* et *Mac* disposent de leur propre interface de fournisseur de service **Spi**, respectivement *CipherSpi*, *SecretKeyFactorySpi*, *KeyGeneratorSpi*, *KeyAgreementSpi* et *MacSpi*.

Parmi les fournisseurs de services cryptographiques qui implémentent JCE, citons ceux qui sont à la fois les plus répandus et les plus complets (tout en étant gratuits !) : *Cryptix JCE* (www.cryptix.org) et *Bouncy Castle* (www.bouncycastle.org), deux produits « open source » qui offrent de nombreux algorithmes supplémentaires par rapport à la version Sun.

Le tableau ci-dessous reprend les principaux algorithmes fournis par les JCE de Sun, Cryptix et Bouncy Castle. Cette liste n'est toutefois ni exhaustive ni définitive, et est sujette à évolution.

FIG. 16 – TABLEAU : PRINCIPAUX ALGORITHMES JCE DE SUN, CRYPTIX ET BOUNCY CASTLE.

Algorithme	***Sun***	***B-Castle***	***Cryptix***	**Technique de sécurité concernée**
Blowfish	✓	✓	✓	Chiffrement
DES	✓	✓	✓	Chiffrement
DESede (TripleDES)	✓	✓	✓	Chiffrement
CAST5			✓	Chiffrement
IDEA		✓	✓	Chiffrement
Kerberos	✓	?	?	Chiffrement & Authentification
MARS			✓	Chiffrement
RC2		✓	✓	Chiffrement
RC4		✓	✓	Chiffrement
RC5		✓		Chiffrement
RC6		✓	✓	Chiffrement
Rijndael (AES)		✓	✓	Chiffrement
Serpent		✓	✓	Chiffrement
Skipjack		✓	✓	Chiffrement
Square			✓	Chiffrement
Twofish		✓	✓	Chiffrement
RSA	✓	✓	✓	Chiffrement asymétrique
ElGamal			✓	Chiffrement asymétrique
Diffie-Hellman	✓	✓	✓	Echange de clés
MD2		✓	✓	Condensé de messages (hachage)
MD4		✓	✓	Condensé de messages (hachage)
MD5	✓	✓	✓	Condensé de messages (hachage)
RipeMD-128		✓	✓	Condensé de messages (hachage)
RipeMD-160		✓	✓	Condensé de messages (hachage)
SHA-0			✓	Condensé de messages (hachage)
SHA-1	✓	✓	✓	Condensé de messages (hachage)
Tiger		✓	✓	Condensé de messages (hachage)
Hmac-MD2		✓	✓	MAC
Hmac-MD4		✓	✓	MAC
Hmac-MD5	✓	✓	✓	MAC
Hmac-RipeMD-128		✓	✓	MAC
Hmac-RipeMD-160		✓	✓	MAC
Hmac-SHA-0			✓	MAC
Hmac-SHA-1	✓	✓	✓	MAC
Hmac-Tiger		✓	✓	MAC
DSA	✓	✓	✓	Signature numérique
RawDSA		✓		Signature numérique

3.2. ZOOM 1 : CHARGEUR DE CLASSE

3.2.1. PRINCIPE DE CHARGEMENT DE CLASSE

[HORSTMANN 00] • [JAWORSKI 01] • [OAKS 01]

Le code destiné à la machine virtuelle Java est stocké par défaut dans un fichier *.class*, qui contient le code complet d'une classe, avec toutes ses méthodes ; ce code sera interprété selon le jeu d'instructions particulier de la machine cible.

Seules les classes nécessaires à l'exécution d'un programme sont chargées, *dynamiquement* (à la demande), selon le procédé suivant :

- Chargement du contenu du fichier *.class* programme (depuis un disque ou une connexion réseau) ;
- Chargement des fichiers de classes liés aux superclasses et aux variables d'instance (ce processus de chargement des classes dépendantes est appelé **résolution de la classe**) ;
- Exécution de la méthode principale de la classe programme ; par défaut, la première classe chargée exécute une méthode statique *main()* ;
- Chargement des classes requises ou appelées par la méthode principale.

On distinguera plusieurs types de chargeurs de classes. Ainsi, le chargeur de classe principal est le *chargeur de classe d'application standard*, qui permet par défaut le chargement dynamique des classes utilisateur selon le procédé général décrit ci-dessus.

Le chargement des classes systèmes bénéficie d'un chargeur spécifique, appelé *chargeur de classe amorce* ou *primordial*, intégré à la machine virtuelle, et qui ne laisse aucun contrôle possible sur son déroulement (une classe système n'est chargée qu'une seule fois et ne peut être remplacée).

Il est possible de fournir un ou plusieurs chargeurs de classe personnalisés pour le chargement des classes d'application (ou d'applet). Un *chargeur de classe personnalisé* remplace le mécanisme de base pour trouver et charger les classes. Il permet d'intégrer des contrôles de sécurité spécifiques à prendre en compte avant de passer les bytecodes à la machine virtuelle (par exemple : contrôle de paiement d'une licence, vérification de l'origine du code, chargement d'une classe chiffrée, etc.).

Le chargeur de classe d'application standard est représenté par la classe *java.lang.ClassLoader*, qui intègre des interfaces de base pour les différentes opérations du processus de chargement. Tout autre chargeur de classe dérive obligatoirement de *ClassLoader*. Même s'il s'agit d'une classe abstraite, aucune de ses méthodes n'est déclarée abstraite depuis Java 2 (dans les versions précédentes, seule la méthode *loadClass()* était abstraite, mais il n'est plus conseillé de dériver cette méthode).

Si les classes d'application standard sont trouvées en suivant le CLASSPATH de la JVM, un chargeur de classe d'applet détermine quant à lui l'origine de la classe au moyen d'une balise CODEBASE du fichier HTML qui lance l'applet. Avec Java 2, la classe *java.net.URLClassLoader*, qui hérite donc de *ClassLoader*, peut charger des classes au départ du réseau HTTP ou à partir du système de fichiers.

Java 2 inclut un troisième type de chargeur de classe standard, *java.security.SecureClassLoader*, qui offre la possibilité d'associer des classes à des permissions et à une *source de code* (une source de code identifie L'URL et le certificat associé à une classe).

Citons encore *java.rmi.server.RMIClassLoader*, hérité de Java 1.1, et utilisé par les applications de type RMI pour la gestion des classes transmises en paramètres ou en valeur de retour.

3.2.2. PROCÉDURE D'UN CHARGEUR DE CLASSE

[HORSTMANN 00] • [JAWORSKI 01]

Principe de base : lorsqu'une classe est chargée par un chargeur de classe particulier, toutes les classes dépendantes sont également chargées par le même chargeur de classe.

Un chargeur de classe donné ne consultera que le chargeur de classe primordial, sa propre cache des classes déjà chargées, et le flux de bytes en entrée. Chaque chargeur de classe possède donc son propre **espace de noms unique**, fonction de l'*origine* de la classe principale : plusieurs chargeurs de classe (s'ils ne sont pas apparentés ; voir section suivante) dans une même machine virtuelle restent indépendants, permettant à des classes différentes mais de même nom pleinement qualifié d'exister dans des chargeurs de classe différents. Un chargeur de classe ne peut donc corrompre un autre chargeur de classe de la même machine virtuelle.

Remarquons que s'il existe plusieurs origines de code, et par là même plusieurs chargeurs de classe, il n'est pas nécessaire d'avoir plusieurs classes différentes, il doit simplement exister une **instance** différente pour chaque origine de code.

Un chargeur de classe est une implémentation de la classe abstraite *ClassLoader*, dont la méthode abstraite *loadClass()* détermine le chargement de la classe de niveau le plus élevé. Avant Java 2, personnaliser un chargeur de classe se résumait à surcharger la méthode *loadClass(String className, boolean resolve)*. Même si cette technique n'est plus conseillée avec Java 2 (nous verrons dans la section suivante quelle est la technique adéquate), il est intéressant d'analyser la procédure suivie.

Cette méthode devait en effet respecter des contraintes d'écriture strictes, dans l'ordre :
- Arrêter le chargement et accepter la classe si elle a déjà été chargée par ce chargeur de classe ;
- S'il s'agit d'une nouvelle classe, arrêter le chargement s'il s'agit d'une classe système (appel au chargeur de classe primordial);
- Charger les bytecodes, au départ du système de fichiers local ou d'une connexion réseau ;
- Définir la classe (i.e. présenter les bytecodes à la machine virtuelle) via la méthode *defineClass()* ;
- Si l'indicateur *resolve* est activé, procéder à la résolution de la classe par appel à la méthode *resolveClass()*. Le chargeur de classe courant sera à nouveau appelé pour le chargement de toutes les classes dépendantes.
- Renvoyer la classe résolue à l'appelant.

NOTES :
- Pour pouvoir vérifier si une classe a déjà été chargée précédemment, un chargeur de classe utilise typiquement une structure de stockage des références des classes déjà chargées (une « cache », sous la forme d'une table de hachage).
- Le chargement d'une classe peut être demandé avec un indicateur *resolve* désactivé (*false*) dans le cas où il s'agit simplement pour la machine virtuelle de vérifier l'existence d'une classe. Une classe doit toutefois avoir été complètement résolue pour pouvoir créer une instance de cette classe ou pour utiliser une de ses méthodes.

3.2.3. DÉLÉGATION DANS JAVA 2

A partir du JDK 1.2, la méthode *loadClass()* n'est plus abstraite ; elle ne doit donc plus être redéfinie lors de la personnalisation du *ClassLoader*. Elle n'est toutefois pas déclarée *final*, ce qui garantit la comptabilité ascendante avec les implémentations selon la technique JDK 1.1. [JAWORSKI 01]

L'évolution de JDK 1.2 est basée sur un constat : le fonctionnement de la méthode *loadClass()* reste applicable de façon générique : il n'est pas nécessaire de réécrire tout le mécanisme de chargement d'une classe alors que la seule opération visée (et autorisée) est en définitive de personnaliser la *présentation* d'une classe à la machine virtuelle.

Java 1.2 met en place un **mécanisme de délégation** du chargement des classes : un chargeur de classe personnalisé délègue d'abord le chargement à sa classe parent, laquelle peut donc être soit une autre classe personnalisée soit, *in fine*, la classe de chargement principale (chargeur d'applications standard).

Il est important ici de définir clairement la notion de parent, qui est totalement dissociée de celle admise dans un contexte d'héritage objet. En effet, pour une classe *A* donnée, le « parent » au sens de l'héritage est la classe que spécialise *A*; c'est donc avant tout une notion liée à la *classe*. Dans le cadre d'un chargeur de classe, la parenté est définie sur les *instances* de classe, lors de la création de celles-ci. Ainsi, on dira que l'instance $\alpha 1$ de la classe *A* a pour parent l'instance $\beta 1$ de la classe *B* (ou même $\alpha 2$ de la classe *A*). Pour créer le lien de parenté, l'instance $\alpha 1$ doit spécifier, lors de l'appel au constructeur de la classe, que son parent est $\beta 1$, par un appel de la forme $A\ \alpha 1 = new\ A(\ \beta 1\ )$.

Par soucis de clarté, nous utiliserons dès maintenant, sauf mention explicite, le terme de **parent** au sens de la parenté des instances, et le terme **ascendant** au sens de la parenté d'héritage de classe.

Le principe de la délégation est simple en soit : *un chargeur de classe commence toujours par demander à son instance parent de charger la classe à sa place*. Ce n'est qu'en cas d'échec du parent (et, par récursivité, après l'échec de toute la parenté de l'instance) que le chargeur de classe courant est autorisé à essayer de charger lui-même la classe.

Si le chargement échoue, le chargeur de classe cède le contrôle à sa méthode *findClass()*, qui permet de spécifier la logique particulière de chargement de classe. C'est donc cette méthode qui doit être dérivée pour créer un chargeur de classe personnalisé. Par défaut, la méthode *findClass()* de *ClassLoader* lance une *ClassNotFoundException*.

3.2.4. ILLUSTRATION DU MÉCANISME DE DÉLÉGATION

Illustrons le mécanisme de la délégation à l'aide d'une situation d'exemple.

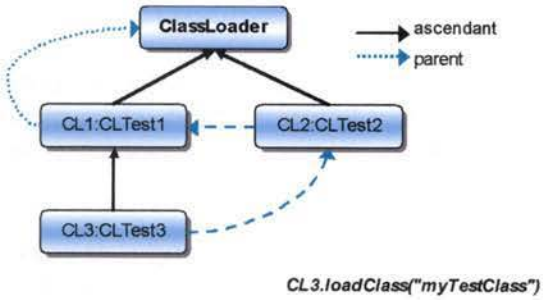


FIG. 17 – DÉLÉGATION DANS UN CHARGEUR DE CLASSE

Soit les classes CLTest1 et CLTest2 dérivées de *ClassLoader*, et une troisième classe CLTest3 dérivée de CLTest1.

Les classes CLTest*x* ont la simple particularité de charger des classes d'extension *.impl* dans un répertoire fixe. Ce répertoire est spécifié via le constructeur; il est nommé *store* par défaut.

CL1, CL2 et CL3 sont des instances respectives de CLTest1, CLTest2 et CLTes3. CL1 est créé à l'aide du constructeur sans paramètre (pas de parent explicite, répertoire store *par* défaut). CL2 est créé en indiquant son parent CL1 et le répertoire *store2* ; CL3 indique son parent CL2 et le répertoire *store3*.

Instance	Classe	Ascendant	Parent	Répertoire
CL1	CLTest1	ClassLoader	/	store
CL2	CLTest2	ClassLoader	CL1	store2
CL3	CLTest3	CLTest1	CL2	store3

Dans l'exemple, une classe *myTestClass.impl* a été placée dans le répertoire *store3*.

La classe *myTestClass* est requise par un appel au chargeur CL3 : *CL3.loadClass("myTestClass")*. Que se passe-t-il alors ?

CL3 ne dispose pas de sa propre méthode *loadClass*() ; la méthode *loadClass*() appelée est donc celle de son ascendant, soit CL2, lequel ne dispose pas non plus d'une méthode propre. C'est donc finalement la méthode directement héritée de *ClassLoader* qui est exécutée.

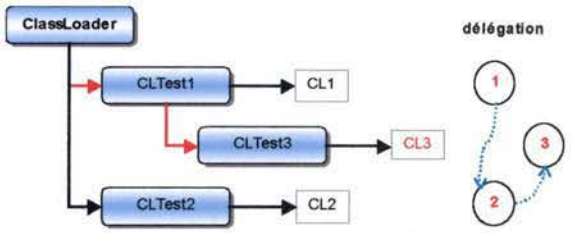


FIG. 18 – ORDRE DE DÉLÉGATION DANS L'EXEMPLE

ClassLoader.loadClass() commence par vérifier si la classe *myTestClass* a déjà été chargée précédemment par l'instance de chargeur de classe courant, soit CL3. Si ce n'est pas le cas, *loadClass()* fait intervenir le chargeur de classe parent : si l'instance courante a un chargeur de classe parent, c'est la méthode *loadClass()* de celui-ci est appelée. De cette manière, **le chargeur de classe délègue toujours d'abord le chargement d'une classe à son parent, avant d'essayer de la charger lui-même si son parent a échoué**. Il faut souligner que ceci est une **définition récursive**, le parent demandant lui-même à son parent éventuel de charger la classe, et ainsi de suite selon la filiation.

Si l'instance courante n'a pas de chargeur parent (ce qui est forcément le cas de l'instance la plus « haute » dans la filiation ; dans l'exemple CL1) , c'est le chargeur de classe par défaut de la machine virtuelle Java qui est appelé par *findBootstrapClass()*. Le chargeur par défaut vérifie d'abord si la classe demandée est une classe système. Sinon, il recherche la classe en parcourant le CLASSPATH.

Si aucun chargeur de classe, ascendant ou système, n'a pu charger la classe demandée, une exception de type *ClassNotFoundException* est finalement générée. La suite du chargement de classe est alors déléguée à la méthode *findClass()*, chargée de décrire le processus particulier de chargement de classe de chaque chargeur personnalisé. Si *findClass()* ne peut résoudre le chargement, l'exception *ClassNotFoundException* est propagée.

Dans l'exemple, c'est donc la méthode *findClass()* de l'instance parent la plus « haute » dans la filiation qui sera appelée en premier, soit *CL1.findClass()*. Si CL1 n'a pas pu charger la classe visée, la méthode de l'instance dont CL1 est le parent est ensuite appelée, soit *CL2.findClass()*, et ainsi de suite jusqu'au renvoi à l'instance de départ, soit *CL3.findClass()*. Ici, CL1 et CL2 échoueront forcément (puisque *myTestClass.impl* a été placé dans le répertoire *store3* auquel est lié CL3), et c'est CL3 qui pourra charger la classe.

La méthode *findClass*() qui trouve la classe à charger lit la suite de bytes correspondant (simple lecture de flux en entrée) et la stocke sous la forme d'un tableau byte[ ].

La méthode *findClass*() présente alors la classe à la machine virtuelle grâce à la méthode *defineClass*(), et retourne la classe *loadClass()* de l'appelant.

La seule action que peut encore entreprendre la méthode *loadClass()* est de demander la résolution de la classe, en fonction du paramètre passé au chargeur de classe (*false* par défaut).

3.2.5. EXEMPLE DE CHARGEUR DE CLASSE PERSONNALISÉ

A titre d'illustration, le code d'une méthode *findClass()* est présenté dans l'encadrement ci-après, telle qu'il pourrait figurer par exemple dans la classe *CLTest1*.

FIG. 19 – CODE SOURCE : EXEMPLE DE CHARGEUR DE CLASSE PERSONNALISÉ

MÉTHODE *CLTEST1.FINDCLASS()*

```
//
// Méthode de chargement de classe invoquée par délégation
// Pré : 'classPath' = variable d'instance contenant le chemin d'accès
// In  : 'name' = le nom de la classe à charger
// Ou  : un objet Class construit au départ de la classe 'name'
//
protected Class findClass(String name) throws ClassNotFoundException
{
  FileInputStream fi = null;

  try
  {
    String path = name.replace('.', '/');

    fi = new FileInputStream(classPath+"/" + path + ".impl");

    byte[] classBytes = new byte[fi.available()];
    fi.read(classBytes);

    return defineClass(name, classBytes, 0, classBytes.length);
  }
  catch (Exception e)
  {
    // Classe non trouvée : propagation de l'exception appropriée
    throw new ClassNotFoundException(name);
  }
  finally
  {
    if ( null != fi ) {
      try {
        fi.close();
      }
      catch (Exception e) { }
    }
  }
}
```

3.2.6. SECURECLASSLOADER ET URLCLASSLOADER

[HORSTMANN 00] • [OAKS 01]

Java 2 propose une classe java.security.***SecureClassLoader*** qui spécialise la classe abstraite *ClassLoader*, en permettant la création de chargeur de classe associé à une origine de code et lié à des permissions d'accès. Si un Gestionnaire de Sécurité a été déclaré, *SecureClassLoader* le prend en compte immédiatement (en commençant par vérifier si la création d'un chargeur de classe est autorisée !). *SecureClassLoader* hérite bien entendu de toute la mécanique relative à la délégation.

Même si elle n'est pas abstraite, *SecureClassLoader* ne peut être utilisée directement, mais doit être dérivée pour obtenir un chargeur de classe opérationnel. En effet, *SecureClassLoader* ne définit pas la manière de charger le tableau de bytecodes. Ainsi, le *ClassLoader* de base ne doit normalement pas être dérivé directement, mais doit être considéré comme *private* (ce qu'il serait sans doute s'il n'y avait pas certaines exigences de compatibilité ascendante entre les versions du JDK).

Après le chargement des bytecodes (effectivement résolu par l'implémentation de la classe qui doit dériver *SecureClassLoader*), un domaine de protection est créé pour la classe, soit en fonction du modèle de sécurité par défaut (fichiers de permissions), soit selon une implémentation particulière fournie lors de la spécialisation du chargeur de classe lui-même. Il est possible également de définir une source de code pour cette classe, et de différer la définition du domaine de protection, qui pourra être obtenu via une méthode *getPermissions()*. Pour plus de détails concernant les sources de code et les domaines de protection, voir *3.3. Zoom 2 : Gestionnaire de sécurité et contrôleur d'accès*.

SecureClassLoader fournit les fonctionnalités indispensables à la gestion de la sécurité et est donc la classe de référence pour toute implémentation d'un chargeur de classe. Ainsi, la classe java.net.***URLClassLoader*** spécialise *SecureClassLoader* en reliant le chargeur de classe sécurisé à une URL particulière pour l'obtention des bytecodes. Cette URL (*Uniform Resource Locator*) représente aussi bien un chemin d'accès du système de fichiers local qu'une référence réseau à un fichier distant.

La classe sun.applet.*AppletClassLoader* permet de charger une applet Java ; depuis JDK 1.2 elle hérite de *URLClassLoader* (elle était auparavant une version concrète directement héritée de *ClassLoader*). Par défaut, lors du chargement de la première classe d'une application, c'est une instance de *URLClassLoader* qui est utilisée ; c'est toutefois une instance de *AppletClassLoader* qui est utilisée s'il s'agit de la première classe d'une applet.

3.2.7. RMICLASSLOADER

[OAKS 01]

La classe particulière appelée java.rmi.server.***RMIClassLoader***, contrairement à ce que son nom indique, n'est pas un véritable chargeur de classe, ni même attaché spécifiquement à RMI. En réalité, la méthode *loadClass*() de *RMIClassLoader* utilise un chargeur de classe « interne », qui est un dérivé de *URLClassLoader*. Sa particularité est qu'il utilise l'URL spécifiée dans la propriété *java.rmi.server.codebase*. En outre, il utilise le même mécanisme de permissions que le chargeur de classe URL standard. Ce chargeur de classe peut en fait être utilisé dans toute application, qu'elle implémente RMI ou non. Son intérêt est davantage historique que technique.

3.3. ZOOM 2 : GESTIONNAIRE DE SÉCURITÉ ET CONTRÔLEUR D'ACCÈS

[GARMS 01] • [GONG 98A] • [HORSTMANN 00] • [JAWORSKI 01] • [OAKS 01] • [SUN 02]

L'implémentation du contrôle d'accès et de la sécurité dans une application Java repose sur trois piliers :

- Le **gestionnaire de sécurité** (*Security Manager*) qui fournit le mécanisme que l'API Java utilise pour voir si une opération relative à la sécurité est autorisée
- Le **contrôleur d'accès** (*Access Controller*) qui fournit la base de l'implémentation par défaut du Security Manager
- Le **chargeur de classes** (*Class Loader*), et plus particulièrement la classe spécialisée *SecureClassLoader*, qui encapsule l'information concernant les classes et les stratégies de sécurité.

Le contrôleur d'accès et le gestionnaire de sécurité poursuivent un but commun : déterminer si une opération particulière doit être autorisée ou refusée. La classe *AccessController* a été ajoutée dans Java 2 pour offrir la fonctionnalité de contrôle d'accès fin et configurable, tandis que la classe *SecurityManager* est conservée pour des raisons de compatibilité avec les applications existantes.

Pourtant l'introduction du contrôleur d'accès ne remplace pas le gestionnaire de sécurité, il le complète et le généralise comme le montre la figure suivante :

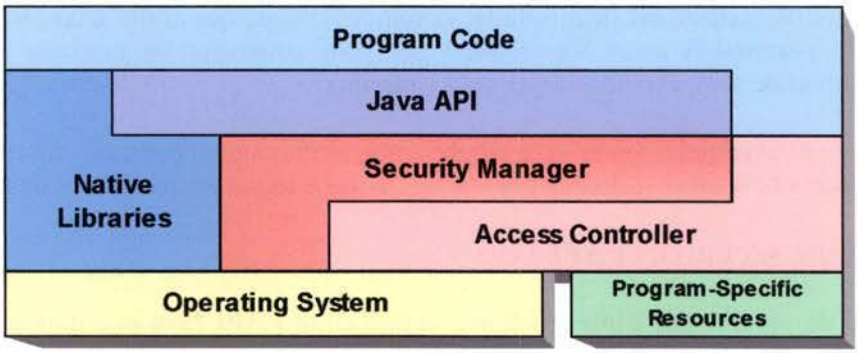


FIG. 20 – COORDINATION ENTRE GESTIONNAIRE DE SÉCURITÉ ET CONTRÔLEUR D'ACCÈS

Ainsi, par exemple, une opération (correspondant à l'exécution d'un programme donné) utilise l'API Java, et atteint les ressources du système en passant successivement par le Security Manager, l'Access Controller et l'Operating System.

Néanmoins, il faut remarquer que, dans certains cas, le gestionnaire de sécurité peut outrepasser le contrôleur d'accès. En effet, dans Java 1.1, il fallait étendre l'objet *SecurityManager* pour obtenir des stratégies de contrôle d'accès personnalisées. Le gestionnaire de sécurité personnalisé ainsi obtenu n'est pas tenu d'utiliser systématiquement le contrôleur d'accès.

Dans Java 2, il est possible de créer de nouvelles permissions en étendant la classe *Permission*. Ces permissions permettent de gérer les ressources relatives aux objets créés par un programme spécifique comme nous le verrons plus loin dans la description du contrôleur d'accès.

Remarquons également que les méthodes natives, généralement écrites en C, ne passent ni par le gestionnaire de sécurité, ni par le contrôleur d'accès. L'utilisation de ces librairies va à l'encontre de la portabilité du langage Java, puisqu'elles ne sont écrites que pour un système d'exploitation particulier. Il faut cependant préciser que le chargement même de ces librairies peut être restreint.

3.3.1. SECURITY MANAGER

Le **gestionnaire de sécurité** est situé au cœur de l'architecture de sécurité de Java : si un programme Java essaye d'ouvrir un fichier, de se connecter à une machine particulière sur le réseau, ou encore de modifier l'état d'un thread, il doit d'abord demander au gestionnaire de sécurité si cette opération lui est autorisée. Si l'accès à la ressource est refusé, une exception est levée.

Ce mode de fonctionnement pourrait laisser croire que Java ne peut être **que** sécurisé. Mais il ne faut pas perdre de vue que si les applets chargent par défaut un gestionnaire de sécurité très restrictif, les applications Java, quant à elles, n'ont aucun gestionnaire de sécurité par défaut.

Pour les **applications** Java, l'utilisation d'un gestionnaire de sécurité doit donc être explicitement spécifiée, soit au démarrage, sur la ligne de commande :

```
Java - Djava.security.manager=default MyApplication
```

soit par programmation, par l'appel de la méthode *setSecurityManager(SecurityManager sm)*.

Ces deux techniques sont tout à fait équivalentes : lorsque l'option est spécifiée sur la ligne de commande, la machine virtuelle exécute la méthode *setSecurityManager()* avant d'appeler la méthode *main()* de l'application.

Il ne peut exister, à un moment donné, qu'**un seul** gestionnaire de sécurité dans une machine virtuelle. Lorsqu'il n'en existe aucun, il est toujours possible d'en créer un et de le charger ; par contre, dès qu'un gestionnaire de sécurité est déjà installé, la source de code qui désire changer de gestionnaire doit posséder la permission *createSecurityManager* pour instancier un nouveau gestionnaire de sécurité et la permission *setSecurityManager* pour l'installer.

Ce mécanisme est essentiel ; sinon une applet mal intentionnée pourrait installer son propre gestionnaire de sécurité et ainsi se donner elle-même l'accès à toutes les ressources de la machine.

GESTIONNAIRE DE SÉCURITÉ ET API JAVA

Le gestionnaire de sécurité est l'**intermédiaire** obligé entre l'API Java et l'implémentation d'une application Java ou d'un navigateur supportant Java. La classe *SecurityManager* de l'API Java fournit l'interface que le reste de l'API Java utilise pour vérifier qu'une opération particulière est permise.

L'algorithme utilisé par l'API Java est toujours le même :

1. Le programmeur utilise une des fonctions de l'API Java pour exécuter une opération.
2. L'API Java demande au gestionnaire de sécurité si l'opération peut être autorisée.
3. Si le gestionnaire de sécurité n'autorise pas l'opération, il retourne une exception *java.langSecurityException* à l'API qui la transmet à l'utilisateur.
4. Sinon, l'API Java exécute l'opération et le programme continue normalement.

Par exemple, lorsqu'un programmeur utilise un objet *FileInputStream* pour lire un fichier, l'API Java construit l'objet en utilisant un code de la forme :

```
Public FileInputStream(String name) throws FileNotFoundExeception {
  SecurityManager sm = System.getSecurityManager() ;
  If (sm != null) {
    sm.checkRead(name);
  }
  try {
    open(name);       // open() est une méthode privée
  } catch (IOException e) {
```

```
        throw new FileNotFoundException(name) ;
    }
}
```

Pour que le gestionnaire de sécurité soit incontournable, il est essentiel que la méthode qui exécute réellement l'opération soit une méthode **privée** de cette classe (elle ne peut donc pas être appelée directement en dehors de la classe où elle a été définie).

Le code du gestionnaire de sécurité est responsable de la décision de lire ou ne pas lire le fichier et, dans ce dernier cas, de lever une exception de sécurité :

```
Public class SecurityManagerImpl extends SecurityManager {
  Public void checkRead(String s) {
    If (theFileIsNotAllowedToBeRead)
        Throw new SecurityException("Not allowed to read"  + s) ;
  }
}
```

La classe *SecurityException* est dérivée de la classe *RuntimeException*. C'est pourquoi elle conduit à l'arrêt du programme si elle n'est pas capturée.

Il est important de noter que l'API Java ne mémorise pas les exceptions déclenchées par les actions qu'elle exécute. Ainsi, même pour une opération identique à une opération précédemment autorisée, l'API Java fera **systématiquement** appel au gestionnaire de sécurité. En effet, nous verrons plus loin (contrôleur d'accès) comment le contexte relatif à une opération est pris en compte : en fonction de ce contexte, une même opération pourra être autorisée dans certains cas et refusée dans d'autres.

MÉTHODES DU GESTIONNAIRE DE SÉCURITÉ

Après avoir étudié le fonctionnement du gestionnaire de sécurité, nous allons voir quelles sont les protections qu'il fournit. La classe *SecurityManager* est principalement constituée d'un ensemble de méthodes publiques de la forme *checkXXX()*. Chacune de ces méthodes vérifie si l'accès est autorisé sur une ressource particulière. En réalité, chaque méthode *checkXXX()* appelle la méthode générique *checkPermission()* qui, à son tour, appelle la classe *AccessController* qui sera décrite dans la section suivante.

Ces méthodes de la forme *checkXXX()* peuvent être classées de la manière suivante :

- Méthodes relatives à l'accès aux fichiers : *checkRead()*, *checkWrite()*, *checkDelete()*
- Méthodes relatives à l'accès au réseau : *checkConnect()*, *checkListen()*, ...
- Méthodes protégeant la Machine Virtuelle Java : *checkCreateClassLoader()*, *checkExec()*, *checkExit()*, ...
- Méthodes protégeant les Threads : *checkAccess()*, ...
- Méthodes protégeant les ressources système : *checkPropertiesAccess()*, ...
- Méthodes protégeant les aspects de la sécurité : *checkSecurityAccess()*, ...

Dans Java 1.1, il fallait étendre l'objet *SecurityManager* pour obtenir des stratégies de contrôle d'accès personnalisées et créer de nouvelles méthodes *checkXXX()*. La plateforme Java 2 décourage ce sous-classement et l'implémentation par défaut du gestionnaire de sécurité le confine aux fonctions de l'API Java.

3.3.2. ACCESS CONTROLLER

Le contrôleur d'accès est construit sur les 4 concepts suivants :

- ***Sources de code*** : une encapsulation de l'origine du code (système de fichiers local, fichier `.jar` ou URL) et d'une liste de signataires (qui peut être vide)
- ***Permission*** : une encapsulation d'une demande d'accès à une ressource particulière
- ***Règles de sécurité (Policies)*** : une encapsulation de l'ensemble des permissions qui peuvent être accordées à des sources de code spécifiques
- ***Domaines de protection*** : une encapsulation d'une source de code et des permissions accordées à cette source de code.

Source de code

Le code Java peut être téléchargé à partir de n'importe où sur un réseau. L'origine du code et son auteur sont donc critiques pour maintenir un environnement de sécurité.

Un code se signe en deux étapes : les fichiers de classe sont d'abord archivés dans un fichier JAR à l'aide de l'outil `jar` ; puis, l'outil `jarsigner` est utilisé par signer le fichier `.jar`. Cet outil ajoute deux fichiers au fichier d'archive : un fichier de signature et un bloc de signature.

Le fichier de signature identifie chaque fichier du fichier `.jar`. Le fichier de bloc de signature contient la signature du fichier de signature et un certificat authentifiant la clé publique du signataire. Un fichier `.jar` peut posséder plusieurs signataires, chacun signant le fichier `.jar` à son tour.

C'est le critère sur lequel se base Java pour accorder ou refuser l'accès aux ressources du système. La philosophie de base est la suivante : **l'utilisateur a la possibilité d'accorder des permissions en fonction de la confiance qu'il place dans une source de code**.

Permission

L'objet *Permission* peut être utilisé dans deux cas différents :

- pour **définir** le domaine de protection d'une classe qui associe à la classe sa source de code et les permissions qui lui ont été accordées
- pour **demander** si une permission particulière peut être accordée (méthodes *checkXXX()* du gestionnaire de sécurité)

Précisons d'emblée que n'importe quelle source de code peut créer n'importe quel objet permission. Mais, détenir un objet permission ne signifie pas posséder les droits d'accès correspondants. Ces droits d'accès sont accordés par le mécanisme des stratégies de sécurité (policies) et des domaines de protections que nous détaillerons dans la section suivante.

Les permissions ont trois propriétés : un **type**, un **nom de ressource** et une **liste d'actions**. Le nom et la liste d'actions sont optionnels. Le type est celui de la classe Java correspondante (par exemple *java.io.FilePermission*). Le nom des permissions de l'API Java sont définis par convention : ainsi, pour un objet *FilePermission*, on utilise le nom du fichier, pour un objet *SocketPermission*, le nom de domaine ou l'adresse IP.

La présence d'une liste d'actions dépend de la sémantique ou du type particulier de la permission : par exemple, pour un objet *FilePermission*, la liste d'actions inclura « read », « write » et « delete ».

Notons que pour une permission de l'API, cette liste n'est pas **extensible** : il n'est donc pas possible de définir un objet représentant la permission de copier un fichier. Pour une permission personnalisée, le programmeur a le champ libre : par exemple, un objet *PayrollPermission* pourrait avoir une liste d'actions reprenant « view » et « update ».

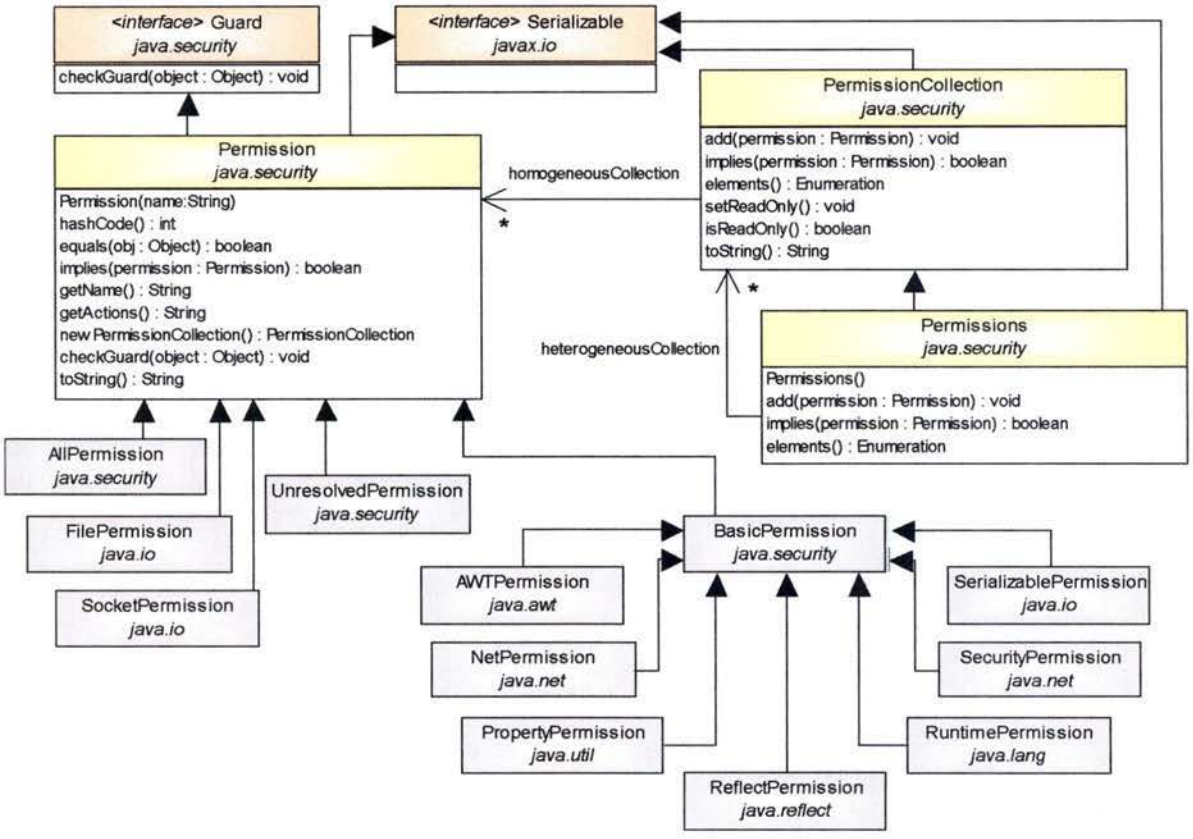


FIG. 21 – ARCHITECTURE DES PERMISSIONS JAVA 2

La figure ci-dessus montre l'**architecture** de base des permissions Java 2. A la racine se trouve la classe abstraite *java.security.Permission*, qui peut être étendue pour créer des permissions personnalisées. Relevons pour cette classe, plusieurs méthodes publiques intéressantes . Les trois premières seront directement utilisées par le contrôleur d'accès. Ces méthodes sont abstraites et doivent être implémentées :

- *public abstract int hashCode()* : pour une permission donnée, l'entier renvoyé par cette méthode doit rester identique durant l'exécution de la machine virtuelle. De plus, des permissions considérées comme égales doivent renvoyer la même valeur.
- *public abstract boolean equals(Permission p)* : permet de déterminer quand deux objets Permission sont égaux, généralement en vérifiant le type, le nom et les actions (s'il en existe) de ces objets.
- *public abstract boolean implies(Permission p)* : cette méthode est une des clés de voûte de la classe Permission. Elle permet de **déterminer si on peut accorder une permission particulière lorsque qu'une permission plus générale a déjà été accordée**. Cette méthode doit tenir compte des caractères génériques. Par exemple une permission de lecture sur le répertoire `/mydir/*` implique une permission de lecture sur le fichier `/mydir/myfile.txt`.
- *public void checkGuard(Object o)* : appelle le gestionnaire de sécurité pour voir si la permission en cours (la variable *this*) a été accordée, générant une *SecurityException* dans le cas contraire. Cette méthode correspond à l'implémentation de l'interface *java.security.Guard* (voir *3.5. Zoom 4 : Signing, Sealing & Guarding Objects*).

La classe *Permission* contient plusieurs sous-classes. Toutes ces sous-classes surchargent et concrétisent les méthodes abstraites définies dans la classe *Permission*.

Un autre moyen de créer sa propre classe Permission est d'étendre la classe *BasicPermission*. Celle-ci permet d'implémenter une permission sans action, de type binaire : quelque chose que l'on a ou que l'on n'a pas. Notons néanmoins que, malgré cette restriction, la classe *PropertyPermission*, dérivée de *BasicPermission* possède les actions « read » et « write ». Il est donc permis d'en faire autant dans une sous-classe personnalisée.

L'intérêt principal de la classe *BasicPermission* est la manière utilisée pour implémenter les caractères génériques dans la méthode *implies(Permission p)*. Les noms des permissions de base suivent une hiérarchie de niveaux séparés par des points. L'astérisque permet de remplacer l'ensemble des permissions d'un même niveau. Par exemple, la société XYZ pourrait créer les permissions *xyz.database.read*, *xyz.database.write* : *xyz.database.** signifie que l'on peut à la fois lire et écrire dans la base de données.

Bien que la classe *BasicPermission* soit elle-même abstraite, elle ne contient aucune méthode abstraite. Pour créer une permission personnalisée en dérivant *BasicPermission*, il suffit donc d'implémenter un constructeur pour appeler le constructeur correct de la superclasse car il n'existe pas de constructeur par défaut (constructeur sans argument) dans la classe *BasicPermission*.

Une des difficultés rencontrées par le contrôleur d'accès est d'assembler les permissions de même type pour être capable d'appeler facilement la méthode *implies()* de chacune d'entre elles.

C'est la raison d'être de la classe *PermissionCollection* qui permet de regrouper un **ensemble homogène de permissions**. Il est possible d'ajouter une permission au groupe par la méthode *add(Permission p)*, de consulter les permissions regroupées avec la méthode *elements()* ou encore de vérifier si une permission de la collection implique une permission particulière grâce à la méthode *implies(Permission p)*. Cette dernière méthode permet une implémentation plus efficace que l'appel de la méthode *implies(Permission p)* pour chaque permission de la collection.

L'ajout de permissions à un objet *PermissionCollection* n'est autorisé que s'il n'est pas en lecture seule. L'accès en lecture seule peut être activé par la méthode *setReadOnly()* et est vérifié par la méthode *isReadOnly()*. La mise en lecture seule d'une collection de permissions est **irréversible**, ce qui permet de se protéger de l'ajout malveillant d'une permission étrangère à la collection qui pourrait fausser le mécanisme de la méthode *implies()* de la collection.

Dans le même esprit, les collections de permissions sont généralement implémentées en tant que classes internes ou, tout au plus, en tant que classes privées pour le paquetage dans lequel elles ont été définies.

Chaque classe dérivée de *Permission* doit implémenter la méthode *newPermissionCollection()* qui renvoie un objet *PermissionCollection*.

La classe *BasicPermission* fournit par défaut une implémentation de la méthode *newPermissionCollection()* qui gère la hiérarchie des noms de ressources et les caractères génériques.

L'intérêt d'une collection homogène de permissions est de faciliter l'écriture de la méthode *implies()*. En pratique, les permissions accordées à une source de code ne sont pas homogènes et il est nécessaire de pouvoir regrouper l'ensemble des permissions accordées à un moment donné. Une classe *Permissions*, dérivée de la classe *PermissionCollection*, permet d'effectuer des **groupements**

hétérogènes de collections de permissions. Pour cela, la classe *Permissions* enregistre les permissions de chaque type dans des objets *PermissionCollection* indépendants.

Lorsque la méthode *implies(Permission p)* de l'objet *Permissions* est appelée, elle cherche la collection correspondant à la permission *p* et appelle ensuite la méthode *implies()* de cette collection homogène pour obtenir la réponse correcte.

RÈGLES DE SÉCURITÉ (*POLICIES*)

Comment l'administrateur de sécurité peut-il décrire une stratégie de sécurité, c'est-à-dire spécifier quelles permissions sont accordées aux différentes sources de code ?

C'est la classe abstraite *java.security.Policy* qui encapsule l'ensembles des règles de sécurité ainsi que la relation entre les permissions et les sources de code auxquelles elles sont accordées.

Pour connaître l'ensemble des permissions accordées à une source de code donnée (définie par une URL et une signature éventuelle), il suffit d'appeler la méthode suivante de la classe *Policy* :

Public abstract PermissionCollection getPermissions(CodeSource cs).

Cette méthode renvoie un objet *PermissionCollection* qui contient, ainsi que nous l'avons défini, un ensemble **homogène** des permissions accordées à la source de code *cs*. Toutefois, depuis la version JDK 1.4, l'objet réellement renvoyé est un objet de type *Permissions*, c'est-à-dire un ensemble **hétérogène** de permissions. C'est évidemment beaucoup plus cohérent avec ce qui a été décrit dans le paragraphe précédent. La classe *Permissions* étant dérivée de la classe *PermissionCollection*, cet artifice permet de conserver la compatibilité avec les versions antérieures.

Une implémentation par défaut de la classe *Policy* est fournie par la classe concrète *sun.security.provider.PolicyFile*, qui construit la stratégie de sécurité sur base des informations trouvées dans un fichier `policy`. Il s'agit d'un fichier texte (ASCII) qui possède une entrée `keystore` facultative et une ou plusieurs entrées de permissions accordées à une source de code selon la forme générale suivante :

```
[keystore "URL_keystore", "type_keystore";]
grant [SignedBy "liste de noms" [, CodeBase "URL"]
      [principal principalClass "principalName",]
{
  permission nom_classe_permission ["nom"] [,"actions"]
      [, SignedBy "liste de noms"]
  permission ...
};
grant ...
```

Nous y retrouvons les éléments précédemment décrits : une source de code définie par une URL (`CodeBase`), une signature (`SignedBy`) et un principal éventuel (`principal`), des permissions accordées à cette source de code, définies par un type (*`nom_classe_permission`*), un nom et une liste d'actions. L'entrée `keystore` désigne l'URL et le type de mécanisme de stockage à consulter pour les signatures.

Le répertoire du fichier `policy` est défini dans le fichier :

`[JAVA_HOME]\lib\security\java.security`

mais il est aussi possible de définir son emplacement sur la ligne de commande :

```
java -Djava.security.policy=MyFile.policy MyProgramTest.
```

Un outil en mode graphique, `policytool`, permet de créer et de modifier les fichiers de sécurité. L'ajout de permissions est simplifié grâce à des listes déroulantes de types de permissions, de noms et d'actions.

Si l'on désire définir les règles de sécurité d'une autre manière qu'à partir d'un fichier texte, par exemple à partir d'une base de données, il est assez facile d'écrire une classe *Policy* personnalisée : il suffit d'écrire une sous-classe de la classe *Policy* et d'implémenter la méthode *getPermissions()*.

En pratique, la réécriture d'une classe *Policy* se justifie dès que le nombre de machines en jeu devient important car l'implémentation par défaut, *PolicyFile*, oblige l'administrateur Java à installer et à mettre à jour les fichiers de règles sur chacune des machines concernées.

Il ne peut exister qu'une seule instance de la classe *Policy* dans la machine virtuelle à un moment donné. Mais cette instance peut être remplacée, par programmation, grâce aux deux méthodes suivantes de la classe *Policy* :

- *static Policy getPolicy()*, qui renvoie l'objet *Policy* en cours
- *public static void setPolicy(Policy p)*, qui remplace l'objet *Policy* en cours par l'objet *p*

à condition de posséder les permissions *getPolicy* et *setPolicy*.

Lorsque la nouvelle instance de la classe *Policy* est installée, seules les classes qui seront chargées ultérieurement dépendront de cette nouvelle instance, à condition de les charger avec un nouveau *ClassLoader*.

Il est aussi possible d'utiliser sa propre classe *Policy* en modifiant le fichier `java.security` et en y remplaçant l'entrée `policy.provider=sun.security.provider.PolicyFile` par une entrée personnalisée (par exemple, `policy.provider=fundp.MyPolicy`). De plus, la classe *MyPolicy* doit se trouver dans le boot classpath de la machine virtuelle, ce qui implique de l'intégrer au fichier `rt.jar` ou de l'ajouter à la ligne de commande (`java -Xbootclasspath :...`).

Cette deuxième technique est plus intéressante car elle permet d'appliquer la classe *Policy* personnalisée à toutes les classes, quel que soit le chargeur de classe.

DOMAINES DE PROTECTION

Un domaine de protection permet de regrouper une source de code et l'ensemble des permissions accordées à cette source de code. Par rapport au fichier `policy` que nous avons décrit plus haut, il s'agit du contenu d'une entrée **`grant`**.

Lorsqu'il est associé à une classe, un domaine de protection identifie l'URL à partir duquel la classe en question a été chargée, avec les signatures éventuelles, et fournit la liste des permissions qui lui sont accordées. **Chaque classe de la machine virtuelle est ainsi liée à un et un seul domaine de protection.**

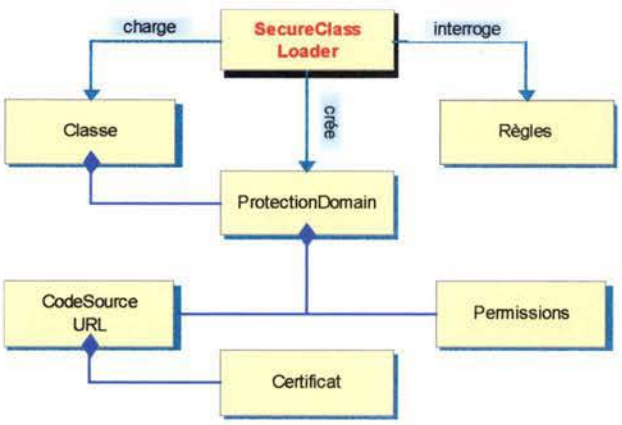


FIG. 22 – DOMAINES DE PROTECTION ET *SECURECLASSLOADER*

Au chargement de la classe, un chargeur de classe héritant de *SecureClassLoader* (ce qui est le cas du chargeur de classe par défaut, *URLClassLoader)* interroge les règles contenues dans la classe *Policy* (ou plus précisément la classe *PolicyFile*, si on conserve le comportement par défaut) pour connaître les permissions accordées à la source de code. Il crée ensuite un objet *ProtectionDomain* avec la

source de code et les permissions données. Le lien entre la classe chargée et le domaine de protection est effectué par un appel à la méthode *private setProtectionDomain(Class, ProtectionDomain)*. Cette méthode privée ne peut être utilisée qu'à l'intérieur du paquetage *java.security*.

L'exemple ci-dessous permet de visualiser un domaine de protection : on y reconnaît l'URL sans signature (dans cet exemple), le chargeur de classe et l'ensemble des permissions associées.

FIG. 23 – CODE SOURCE : EXEMPLE DE DOMAINE DE PROTECTION

Exemple :
```
ProtectionDomain domain = c1.getClass().getProtectionDomain();
System.out.println(domain.toString());
```

```
ProtectionDomain  (file:/C:/Memoire/src/TestPermission/ <no certificates>)
 sun.misc.Launcher$AppClassLoader@209f4e
 <no principals>
 java.security.Permissions@20d62b (
 (java.io.FilePermission \C:\Memoire\src\TestPermission\- read)
 (java.net.SocketPermission localhost:1024- listen,resolve)
  ...
 (java.lang.RuntimePermission exitVM)
 (java.lang.RuntimePermission accessClassInPackage.sun.*)
 (java.lang.RuntimePermission stopThread)
 (java.lang.RuntimePermission getProtectionDomain)
)
```

Lorsqu'une autre classe correspondant à la même source de code est chargée à son tour, elle obtiendra les mêmes permissions et il n'y aura pas besoin de créer de nouveau domaine de protection. Par contre, si une autre classe originaire d'une autre source de code obtient les mêmes permissions, un nouveau domaine de protection sera créé. Une autre source de code ne signifie pas nécessairement une URL différente : il peut s'agir de la même URL avec une signature différente.

Toutes les classes appartenant à un même domaine de protection bénéficient des mêmes permissions. Par contre il est possible que deux classes, bénéficiant des mêmes permissions, appartiennent à deux domaines de protection différents.

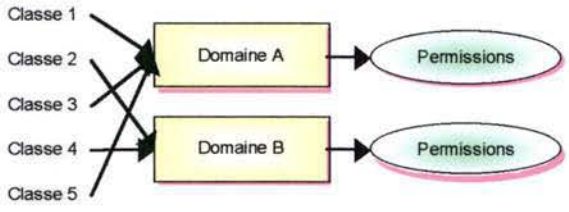


FIG. 24 – PERMISSIONS D'UN DOMAINE DE PROTECTION

Lorsqu'un domaine est créé, l'ensemble des permissions accordées est mis en lecture seule par la méthode *setReadOnly()* définie dans la classe *Permissions*, empêchant ainsi toute modification ultérieure.

Soulignons qu'il existe un domaine de protection particulier, le **domaine système** : toutes les classes de l'API Java, chargées à partir du chargeur de classe système (`null` *classloader*) appartiennent à ce domaine. Ce domaine possède des privilèges spéciaux qui permettent le bon fonctionnement de ces classes systèmes.

LE CONTRÔLEUR D'ACCÈS

A présent, nous disposons de tous les éléments nécessaires à la compréhension des mécanismes du contrôleur d'accès : en fonction de sa source de code d'origine, chaque classe est liée à un domaine de protection qui a été créé au chargement de la classe sur base de l'objet *policy*, lui-même construit à partir d'une « base de données » gérée par l'administrateur Java (fichiers `policy`).

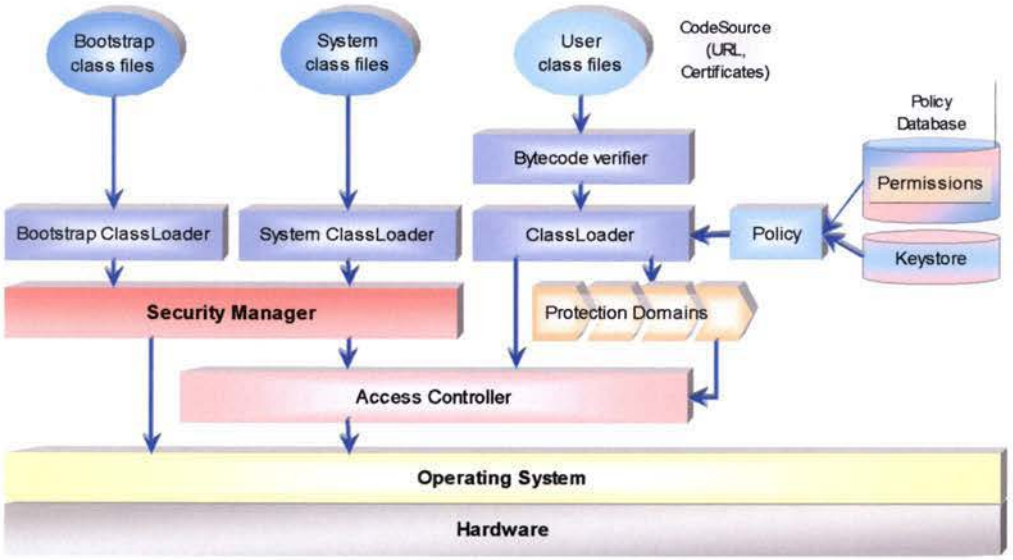


FIG. 25 – LE CONTRÔLEUR D'ACCÈS AU COEUR DE L'ARCHITECTURE DE SÉCURITÉ JAVA 2

Le contrôleur d'accès est représenté par une simple classe *java.security.AccessController*. Il n'existe pas d'instance de cette classe – son constructeur est *private* – et ne peut donc être instancié.

Par contre, cette classe possède une série de méthodes statiques qui peuvent être appelées pour déterminer si une opération particulière peut être effectuée.

La méthode clé de cette classe, ***public static void checkPermission(Permission p)***, permet de déterminer si une permission peut être accordée ou non, en se basant sur l'objet *Policy* installé. Si la permission est accordée, le programme continue normalement, sinon il lève une exception *AccessControlException*.

Pour vérifier si une permission particulière est accordée, le contrôleur d'accès analyse les domaines de protection de **toutes les classes de la pile d'appel de la classe courante**. L'autorisation ne sera accordée que si elle est en concordance avec les domaines de protection de toutes ces classes, c'est-à-dire si la permission appartient à l'**intersection** des domaines de protection des classes de la pile d'appel.

En effet, ainsi que nous l'avons déjà précisé, les classes de l'API Java appartiennent au domaine de protection système, qui possède des privilèges spéciaux (*AllPermission*). Ceci permet à ces classes d'accéder aux ressources système sans en avoir reçu explicitement l'autorisation.

Par exemple, la classe *FileReader* appartient au domaine de protection système qui peut être visualisé comme suit (source de code = `null`, chargeur de classe = `null`) :

FIG. 26 – CODE SOURCE : EXEMPLE DE DOMAINE DE PROTECTION FILEREADER

```
ProtectionDomain  null
 null
 <no principals>
 java.security.Permissions@1503a3 (
 (java.security.AllPermission <all permissions> <all actions>)
)
```

Si la classe *FileReader* ne possédait pas la permission *AllPermission*, et en particulier celle d'ouvrir n'importe quel fichier, elle ne serait pas utilisable. D'autre part, si le contrôleur d'accès ne vérifiait que le domaine de protection de la classe en cours, *FileReader* permettrait à n'importe quelle classe de lire n'importe quel fichier.

Imaginons une applet *ReadFileTest* qui procède à la lecture d'un fichier `test.txt` :

```
import java…

public class ReadFileTest extends Applet {
  public void readTest() {
    try {
      FileReader fileTest = new FileReader("Test.txt");
      System.out.println(new BufferedReader(fileTest).readLine());
      fileTest.close();
    }
    catch (FileNotFoundException fnfe) { fnfe.printStackTrace(); }
    catch (IOException ioe) { ioe.printStackTrace(); }
  }
}
```

La figure ci-contre nous montre l'état de la pile d'appel au moment de l'exécution du constructeur *FileReader*.

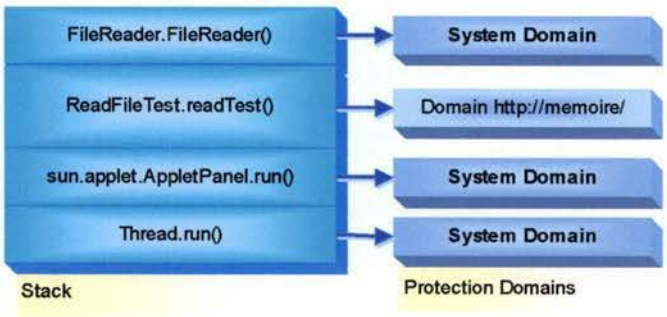


FIG. 27 – EXEMPLE DE PILE D'APPEL : *FILEREADER*

Dans cet exemple, l'intersection des domaines de protection sera le domaine de protection du code source de l'applet, ce qui correspond exactement à ce que l'on désirait dans ce cas.

Ce mécanisme très important empêche donc une classe quelconque d'utiliser les privilèges d'une classe de confiance, telle qu'une classe de l'API Java.

Néanmoins, il existe certains cas où il est nécessaire de contourner cette restriction. Voici un exemple : un administrateur Java écrit une classe d'audit chargée d'écrire dans un fichier `log.txt` toutes les opérations effectuées par une application ainsi que la date et l'heure de chaque opération. Le fichier `log.txt` est enregistré dans le répertoire de l'utilisateur et la classe *Log* reçoit la permission d'écrire ce fichier.

Chaque classe de l'application est obligée d'appeler la méthode *logAction(String text)* de la classe *Log* pour enregistrer les opérations réalisées. Mais lorsqu'une classe de l'application, chargée depuis une

source de code quelconque, exécute cette méthode, le contrôleur d'accès lève une exception de sécurité car cette classe n'est pas autorisée à écrire dans le fichier `log.txt` !

Une solution à ce problème serait d'accorder à toutes les classes de l'application l'autorisation d'écrire le fichier `log.txt`. Mais cette solution n'est guère acceptable du point de vue sécurité, car l'administrateur doit contrôler exactement ce qui sera écrit dans le fichier `log.txt`.

La solution adéquate à ce problème est fournie par la possibilité pour l'administrateur d'écrire sa classe *Log* sous la forme d'un **code privilégié**, c'est-à-dire implémentant une des deux interfaces suivantes : *java.security.PrivilegedAction* ou *java.security.PrivilegedExceptionAction*.

Le choix entre ces deux interfaces se fera en fonction de la nécessité de traiter ou non les exceptions détectées dans le code privilégié. Dans notre cas, il suffira d'implémenter *PrivilegedAction*. Il faut ensuite appeler la méthode ***public static object doPrivileged(PrivilegedAction action)*** de la classe AccessController.

La figure ci-contre nous donne l'état de la pile d'appel au moment de l'appel de la méthode *FileInputStream()* pour accéder au fichier `log.txt`.

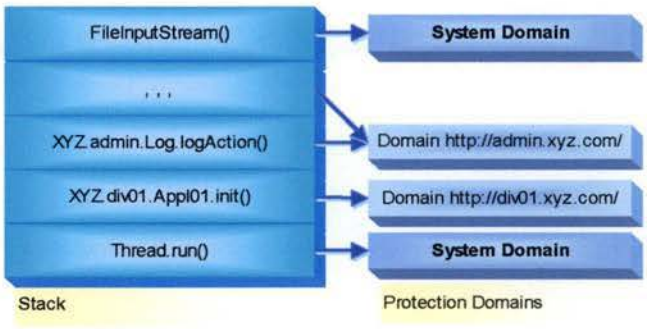


FIG. 28 – EXEMPLE DE PILE D'APPEL : *FILEINPUTSTREAM*

Lorsque le contrôleur d'accès examine les différents domaines de protection liés à l'appel de cette méthode, il arrêtera sa recherche à la méthode privilégiée *Log.logAction* : l'intersection des domaines de protection correspondra alors à l'intersection du domaine système et du domaine de la classe *Log*. Cette dernière ayant reçu la permission d'écrire dans le fichier log.txt, l'écriture dans la fichier sera accordée.

Lorsqu'un code est rendu privilégié, cela ne signifie donc pas qu'il accorde ses propres permissions à la classe appelante. En fait, le code privilégié détermine un point d'arrêt dans la recherche de l'intersection des domaines de protection de la pile d'appel. De ce fait, la classe appelante obtient les permissions de la classe privilégiée uniquement pour la durée de l'appel à cette classe.

Notons aussi qu'un code privilégié ne peut obtenir davantage de permissions que celles qui lui ont été effectivement accordées. Par contre, la classe appelante peut exécuter le code privilégié sans pour autant posséder les permissions nécessaires à l'exécution du code privilégié.

Enfin, si un code privilégié peut accorder ses propres permissions à une classe appelante, il ne lui est pas possible d'accorder ces mêmes permissions à une classe appelée. En effet, le code privilégié n'accorde pas de permissions au sens strict, mais il crée un point d'arrêt dans la pile d'appel et cette pile d'appel est parcourue depuis la classe en cours jusqu'à la classe de lancement de l'application.

3.4. ZOOM 3 : SECURE SOCKET LAYER

[GARMS 01] • [JAWORSKI 01] • [TRAVIS 01]

Nous allons montrer ici la facilité avec laquelle il est possible d'implémenter SSL dans une application Java avec JSSE. Pour ce faire, nous illustrerons les techniques Java au travers d'un exemple simple, extrait d'un tutorial IBM.

3.4.1. ILLUSTRATION : WHITEBOARD

L'application *WhiteBoard* consiste en un tableau blanc distribué. Le tableau blanc est un programme qui permet à un utilisateur d'écrire un message dans une fenêtre. Le texte de ce message apparaît dans toutes les fenêtres des utilisateurs connectés au même serveur de tableau blanc distribué. Ce système permet aux utilisateurs de communiquer entre eux.

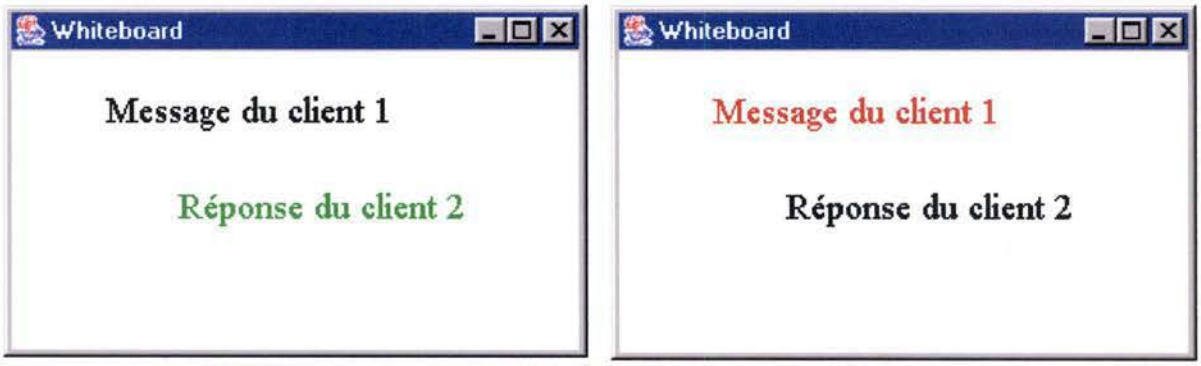


FIG. 29 – ILLUSTRATION SSL : *WHITEBOARD*

Chaque utilisateur voit son propre message en noir tandis que sur le tableau blanc des autres utilisateurs, le texte apparaît en couleur.

Ce système est basé sur une architecture client/serveur. Du côté serveur, il existe une classe *Server* qui écoute les connexions entrantes sur un port spécifique. Pour chaque connexion, le serveur crée un *ConnectionProcessor*. Ce processus est chargé de recevoir les messages et de les transmettre à tous les clients du serveur, comme l'illustre la figure ci-contre.

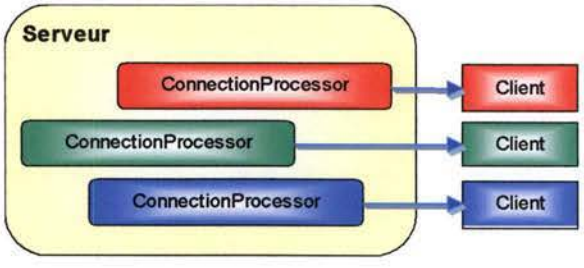


FIG. 30 – ILLUSTRATION SSL : *CONNECTIONPROCESSOR*

3.4.2. GESTION DES CLÉS

Le mécanisme du protocole SSL a été décrit dans la section *2.3.7. SSL – Secure Socket Layer*. Il se divise en deux parties : une authentification réciproque à l'aide de la technique des clés publique/privée et une communication chiffrée à l'aide d'une clé secrète échangée durant le protocole d'établissement de la connexion..

Dans un premier temps, nous devons donc générer les paires clés publique/privée du serveur et des différents clients. Ces clés seront stockées dans un objet **keystore**. Un keystore, ou réserve de clés, est

un conteneurs de clés secrètes, de paires de clés publique/privée et de certificats attestant la validité d'une clé publique. La classe *java.security.Keystore* encapsule la notion d'objet keystore. Il s'agit d'une classe abstraite implémentée en fonction du fournisseur.
L'outil `keytool`, introduit à partir de Java 2, fournit une implémentation de la classe *Keystore*. Il supporte les fonctions suivantes :

- **Génération de paire de clés** : génère une paire de clés publique/privée (c'est-à-dire une entrée de clés) et crée un certificat X.509 autosigné pour la clé publique.
- **Gestion des entrées de clés** : permet de créer, d'importer, d'exporter, de lister et de supprimer les entrées de clés.
- **Gestion des entrées de certificats approuvés** : permet d'importer, d'exporter, de lister et de supprimer les certificats.
- **Génération de requêtes de signature de certificat** : permet de générer des requêtes de signature de certificats (CSR, *Certificate Signing Request*) pour les soumettre à une autorité de certification.
- **Gestion des mots de passe** : les mots de passe peuvent être utilisés pour protéger des clés privées ou la totalité du keystore.

La table ci-dessus décrit les quatre keystore que nous allons utiliser dans notre exemple :

FIG. 31 – TABLEAU : EXEMPLES DE KEYSTORES

Fichier Keystore	Contenu	Localisation
client.private	La paire clés publique/privée du client	Côté client
server.public	Le certificat du serveur (clé publique)	Côté client
server.private	La paire clés publique/privée du serveur	Côté serveur
server.public	Le certificat du client (clé publique)	Côté serveur

Il est important de noter que les clés privées ne peuvent se trouver que du côté de l'entité à laquelle elles appartiennent.

Les commandes suivantes permettent de créer ces quatre fichiers keystore :

FIG. 32 – CODE SOURCE : EXEMPLE DE CRÉATION DE KEYSTORES

```
rem Génère la paire de clés publique/privée du client dans <client.private>
keytool -genkey -alias clientprivate -keystore client.private -storetype JKS -
keyalg rsa -dname "CN=Your Name, OU=Your Organizational Unit, O=Your
Organization, L=Your City, S=Your State, C=Your Country" -storepass clientpw -
keypass clientpw

rem Génère la paire de clés publique/privée du serveur dans <server.private>
keytool -genkey -alias serverprivate -keystore server.private -storetype JKS -
keyalg rsa -dname "CN=Your Name, OU=Your Organizational Unit, O=Your
Organization, L=Your City, S=Your State, C=Your Country" -storepass serverpw -
keypass serverpw

rem Exporte la clé publique du client et l'importe dans <client.public>
keytool -export -alias clientprivate -keystore client.private -file temp.key -
storepass clientpw
keytool -import -noprompt -alias clientpublic -keystore client.public -file
temp.key -storepass public
erase temp.key

rem Exporte la clé publique du serveur et l'importe dans <server.public>
keytool -export -alias serverprivate -keystore server.private -file temp.key -
storepass serverpw
keytool -import -noprompt -alias serverpublic -keystore server.public -file
temp.key -storepass public
erase temp.key
```

3.4.3. COMMUNICATION SSL

Dans notre exemple, nous utilisons des sockets pour réaliser la communication client/serveur. Nous allons utiliser des **sockets sécurisés** destinés à garantir la confidentialité de la communication. Les sockets sécurisés fonctionnent de la même manière que les autres, mis à part qu'ils chiffrent et déchiffrent de manière transparente les données échangées.

Voyons d'abord comme fonctionne une communication réseau en utilisant les sockets de base. Du côté client, il suffit de créer un nouveau *Socket* vers l'ordinateur distant *host* sur le port *port* :

```
Socket socket = new Socket(host, port);
```

Du côté serveur, il faut créer un *ServerSocket* à l'écoute d'un port particulier *port* et ensuite créer une boucle infinie qui accepte et traite les connexions entrantes :

```
ServerSocket serverSocket = new ServerSocket(port);
while (true) {
  Socket socket = serverSocket.accept();
  DoSomethingWithNewConnection(socket);
}
```

Pour pouvoir utiliser les sockets sécurisés, nous devons alors compléter quelques étapes, **côté client** :

1. Créer une source de nombres aléatoires *SecureRandom*. Ces nombres aléatoires seront utilisés durant l'initialisation de l'objet *SSLContext* et doivent être « suffisamment » aléatoires pour éviter les attaques, c'est-à-dire difficiles à reproduire.

   ```
   SecureRandom = new SecureRandom();
   SecureRandom.nextIn(); // Force l'initialisation de l'objet SecureRandom
   ```

2. Créer un objet *Keystore* contenant la clé publique du serveur. Ce qui est réalisé par la lecture du fichier `server.public`.

   ```
   private void setupServerKeystore()
       throws GeneralSecurityException, IOException {
     serverKeyStore = KeyStore.getInstance( "JKS" );
     serverKeyStore.load( new FileInputStream( "server.public" ),
                          "public".toCharArray() );
   }
   ```

3. Créer un objet *Keystore* contenant la paire de clés publique/privée du client. Ce qui est réalisé par la lecture du fichier `client.private`.

   ```
   private void setupClientKeyStore()
       throws GeneralSecurityException, IOException {
     clientKeyStore = KeyStore.getInstance( "JKS" );
     clientKeyStore.load( new FileInputStream( "client.private" ),
                          passphrase.toCharArray() );
   }
   ```

4. Créer un objet *TrustManagerFactory* à partir du *Keystore* du serveur distant. Cet objet sera utilisé pour authentifier le serveur distant. Comme son nom l'indique, cet objet permet de gérer les autorités de certification, et de décider si un certificat peut être accepté ou non .
5. Créer un objet *KeyManagerFactory* à partir du *Keystore* du client. Cet objet sera utilisé pour chiffrer et déchiffrer les données.

6. Créer un objet *SSLContext* en utilisant les objets *TrustManagerFactory*, *KeyManagerFactory* et *SecureRandom*, précédemment créés.

```
private void setupSSLContext()
    throws GeneralSecurityException, IOException {
  TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
  tmf.init( serverKeyStore );

  KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
  kmf.init( clientKeyStore, passphrase.toCharArray() );

  sslContext = SSLContext.getInstance("SSL");
  sslContext.init( kmf.getKeyManagers(),
                   tmf.getTrustManagers(),
                   secureRandom );
}
```

7. Utiliser l'objet *SSLContext* pour créer un objet *SSLSocketFactory*.

```
SSLSocketFactory socketFactory = sslContext.getSocketFactory();
```

8. Utiliser l'objet *SSLSocketFactory* pour créer un objet ***SSLSocket***, qui fonctionnera comme un socket standard, mis à part qu'il est sécurisé.

```
SSLSocket socket = (SSLSocket)socketFactory.createSocket(host, port);
```

Une liste similaire d'étapes doit également être complétée **côté serveur** : il y a peu de différences avec les étapes permettant de créer le socket client.

Les objets *Keystore* contiendront respectivement la clé publique du client (étape 2) et la paire de clés publique/privée du serveur (étape 3). L'objet *TrustManagerFactory* sera créé à partir du K*eystore* du client et l'objet *KeyManagerFactory* à partir du K*eystore* du serveur.

A l'étape 7, le *SSLContext* sera utilisé pour créer un objet *SSLServerSocketFactory* qui sera utilisé à son tour pour créer un objet ***SSLServerSocket***, lequel fonctionnera comme un socket serveur standard mais sécurisé.

Ainsi que nous l'avons vu dans cet exemple, pour construire un socket sécurisé de type SSL, il faut tout d'abord construire un contexte SSL qui contient les clés et les certificats nécessaires à l'établissement d'une connexion SSL. Ce contexte est utilisé pour construire une « fabrique » spécialisée de sockets sécurisés (client ou serveur).

Enfin, les sockets sécurisés ainsi créés s'utilisent de la même manière que les sockets de base, mais les données échangées sont chiffrées au départ et déchiffrées à l'arrivée, de façon tout à fait **transparente**.

Notons encore que les objets *Keystore*, *TrustManagerFactory* et *KeyManagerFactory* fournis en standard ont SUN pour fournisseur et peuvent être remplacés par des objets d'autres fournisseurs.

3.5. ZOOM 4 : OBJETS SIGNÉS, SCELLÉS, GARDÉS

[GARMS 01] • [GONG 98B] • [JAWORSKI 01] • [OAKS 01] • [SUN 02]

Ainsi que nous l'avons souligné dans la section *2.4. Langages Orientés Objet et Sécurité*, la nature "orienté objet" du langage Java introduit des problèmes de sécurité supplémentaires par certains aspects spécifiques tels que l'encapsulation, l'espace de noms ou la diversité des types.

Ces difficultés ont conduit tout naturellement à l'existence dans Java, de mécanismes de protection au niveau des objets eux-mêmes. Il s'agit d'assurer à tout moment la protection de l'état interne à l'exécution (*runtime's internal state*).

Ces mécanismes permettent aux programmeurs :
- d'aller au-delà de la protection naturelle du langage Java
- de gérer l'accès aux ressources à l'intérieur des « threads » (*Guarding Objects*).

Par exemple, dans une application Java distribuée utilisant RMI à travers plusieurs JVM, il est nécessaire de protéger l'**intégrité** et la **confidentialité** des objets. Ces exigences de sécurité existent aussi bien pour des objets situés en mémoire, en transit (stockés dans des paquets IP) ou encore sauvegardés sur disque.

Nous allons décrire les objets signés, scellés et gardés. Ces constructions permettent d'enrichir les mécanismes de sécurité existants de Java et rendent les applications sécurisées plus faciles à construire.

3.5.1. OBJETS JAVA SIGNÉS (SIGNING JAVA OBJECTS)

Un Objet signé, *SignedObject*, contient un autre objet sérialisable, l'objet « à signer », et sa signature. Si la signature n'est pas nulle, il contient une signature digitale valide de l'objet « à signer ».

La signature est un digest de l'objet sérialisé, chiffré à l'aide de la clé **privée** du signataire.

Ce mécanisme permet de créer des objets dont l'**intégrité** est garantie par un tiers (le signataire) et peut être vérifiée à l'aide de sa clé **publique** (voir *1.1.3. Vérification de l'Identité*). Ces objets ne peuvent être falsifiés sans que cela soit décelé, pour autant que l'algorithme de signature soit fiable.

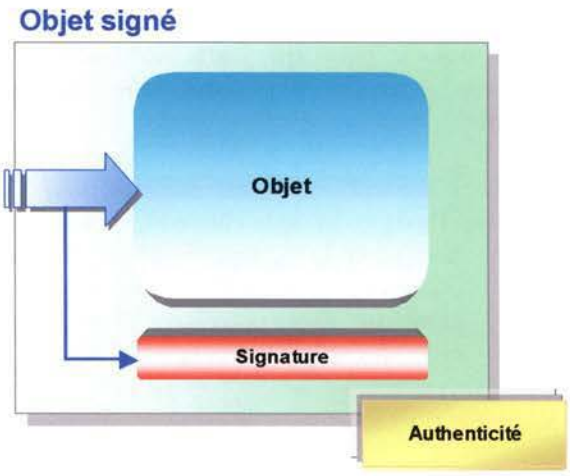


FIG. 33 – OBJET SIGNÉ

Un objet signé est une **copie** parfaite de l'objet original. Soulignons qu'une fois la copie faite, les modifications ultérieures opérées sur l'objet original n'ont plus **aucun effet** sur la copie.

Parmi les différentes applications des objets signés, nous pouvons citer :

- l'utilisation d'un objet signé à l'intérieur d'une machine virtuelle Java en tant que jeton d'autorisation infalsifiable, qui pourrait circuler et ne pourrait être modifié sans que cela soit automatiquement détecté ;

- la transmission fiable d'objets signés entre machines virtuelles Java ;
- le stockage à l'extérieur de Java, ce qui permettrait, par exemple, de redémarrer une machine virtuelle Java à partir d'un état préalablement sauvegardé ;
- l'utilisation d'objets signés « emboîtés » pour construire une suite de signatures, correspondant à une chaîne d'autorisations et de délégations.

Les algorithmes de signature peuvent être, entre autres, les standards DSA (avec l'algorithme de digest SHA-1 : `"SHA-1/DSA"`) et RSA (avec un des algorithmes de digest MD2, MD5 ou SHA-1 : `"MD2/RSA"`, `"MD5/RSA"`, `"SHA-1/RSA"`). Ces algorithmes sont fournis par le provider SUN et peuvent être implémentés par d'autres providers.

Les signatures des méthodes de la classe *SignedObject* sont les suivantes :

- `public SignedObject(Serializable objectToBeSigned, PrivateKey signingKey, Signature signingEngine)`, le constructeur de la classe.
- `public final boolean verify(PublicKey verificationKey, Signature verificationEngine)`, vérifie que la signature contenue dans l'objet est valide.
- `public final String getAlgorithm()`, renvoie le nom de l'algorithme de signature.
- `public final Object getObject()`, renvoie l'objet qui a été signé.
- `public final byte[] getSignature()`, renvoie la signature de l'objet signé sous forme de tableau d'octets.

Notons qu'il n'est pas nécessaire d'initialiser explicitement l'objet *Signature* pour créer ou utiliser un objet signé car il est réinitialisé tant par le constructeur de l'objet signé que par la méthode *verify()*.

D'autre part, il est important de préciser que, pour des raisons de flexibilité, il est possible d'utiliser un objet signature personnalisé, c'est-à-dire créé par un provider reconnu par le système (correspondant à une entrée du fichier `java.security`). Dans ce cas, la méthode *verify()* invoquée pour vérifier la signature de l'objet signé sera celle de l'objet *Signature* personnalisé.

Il est donc crucial que cette méthode renvoie un résultat fiable. En d'autres termes, un objet *Signature* malveillant pourrait renvoyer *true* systématiquement pour outrepasser le contrôle de sécurité.

La classe *SignedObject* a été déclarée *final*. Cette limitation devrait être supprimée dans le futur pour autoriser l'extension de la classe *SignedObjet*, ce qui permettrait entre autres de :

- compléter la signature par un certificat (qui contient le nom du signataire ainsi que sa clé publique)
- signer un même objet avec plusieurs signatures
- créer des objets typés (c'est-à-dire dont le type serait plus spécialisé qu'*Object*).

Exemple d'utilisation :

- Création de l'objet

```
Signature signingEngine = Signature.getInstance(algorithm [, provider]);
SignedObject so = new SignedObject (objectToBeSigned, signingKey, signingEngine);
```

- Utilisation de l'objet reçu

```
String algorithm = so.getAlgorithm();
Signature verificationEngine = Signature.getInstance(algorithm [, provider]);
if (so.verify(publicKey, verificationEngine))
  try {
        Object myObj = so.getObject();
      }
```

```
catch (java.lang.ClassNotFoundException e) {};
```

3.5.2. OBJETS JAVA SCELLÉS (SEALING JAVA OBJECTS)

Un objet scellé, *SealedObject*, contient un autre objet, sérialisable et protégé par un algorithme de chiffrement.

Les objets scellés sont complémentaires des objets signés ; ils garantissent la confidentialité là où les objets signés garantissaient l'intégrité.

En fait, les deux classes auraient pu n'en faire qu'une, mais elles ont été gardées séparées parce qu'elles correspondent à des lois d'exportation américaines différentes.

Pour cette même raison, la classe *SignedObject* se trouve dans le paquetage *java.security* tandis que la classe *SealedObject* est située dans le paquetage *javax.crypto*.

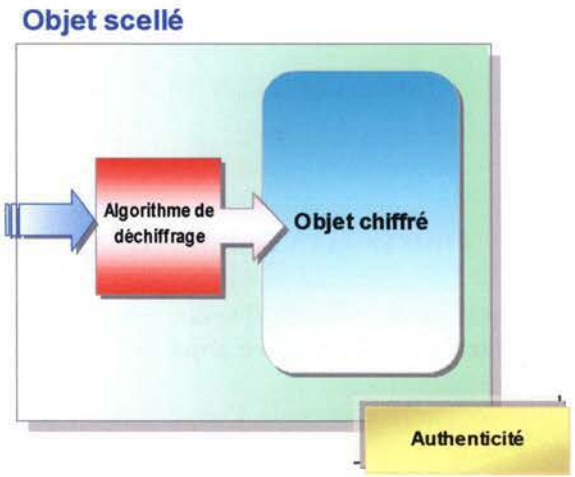


FIG. 34 – OBJET SCELLÉ

Dans la pratique, il est commode d'utiliser un objet scellé et signé. Dans ce cas, il est recommandé de créer d'abord l'objet signé et de l'utiliser ensuite dans un objet scellé, ce qui permet d'éviter de signer aveuglément des données chiffrées.

Contrairement à la classe *SignedObject*, la classe *SealedObject* n'a pas été déclarée *final* et peut donc être étendue. On évite ainsi le casting de l'objet à l'intérieur de la sous-classe, ce qui permet de mieux préserver les informations statiques de typage.

Les signatures des objets scellés sont les suivantes :

- `Public SealedObject(Serializable objectToBeSealed, Cipher c)`, le constructeur de la classe.
- `Public final Object getObject(Cipher c)`, qui permet de récupérer l'objet original à condition de fournir un objet Cipher correctement initialisé.
- `Public final Object getObject(Key key)`, qui permet de récupérer l'objet original en ne fournissant que la clé de scellement.

Exemple d'utilisation :

- Génération d'une clé DES

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES");
SecretKey desKey = keyGen.generateKey();
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, desKey);
```

- Création de l'objet

```
String s = new String("Greetings");
SealedObject so = new SealedObject(s, cipher);
```

- Utilisation de l'objet reçu

```
cipher.init(Cipher.DECRYPT_MODE, desKey);
try {
      String s = (String) so.getObject(cipher);
    }
catch (java.lang.ClassNotFoundException e) {}
```

3.5.3. OBJETS JAVA GARDÉS (GUARDING JAVA OBJECTS)

Un objet gardé, *GuardedObject*, est utilisé **pour protéger l'accès à un autre objet**.

Un objet gardé encapsule l'objet protégé avec un objet implémentant l'interface *Guard*. Cet objet *Guard* invoque la méthode *checkGuard* avec l'objet ressource en paramètre à chaque fois qu'un utilisateur veut se servir de l'objet ressource.

Il lance une exception *SecurityException* si l'accès à la ressource n'est pas autorisé.

Cette nouvelle classe a été créée pour des applications où le fournisseur d'une ressource ne se trouve pas dans le même contexte d'exécution (typiquement un thread différent) que le consommateur de la ressource.

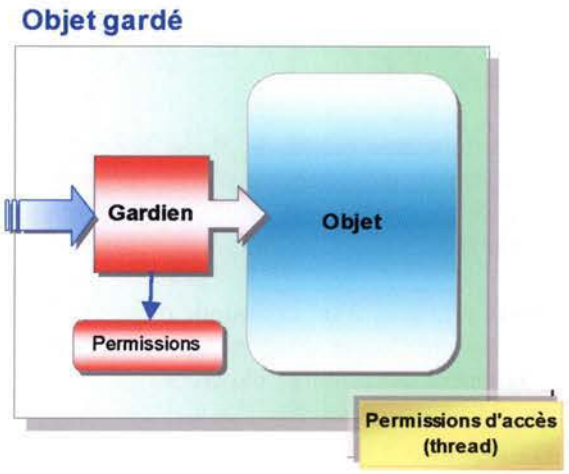


FIG. 35 – OBJET GARDÉ

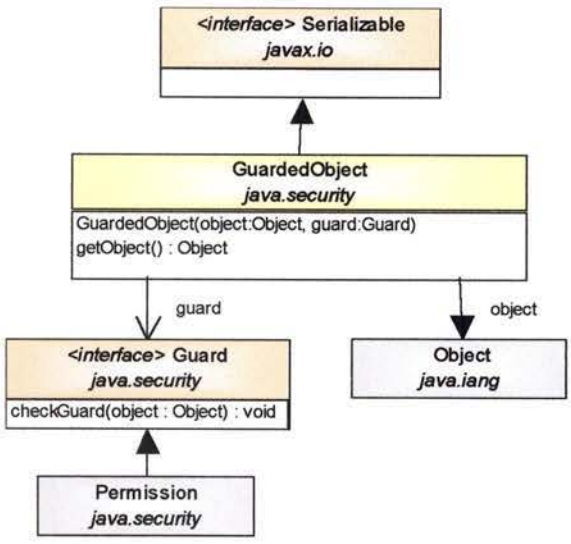


FIG. 36 – LA CLASSE *GUARDEDOBJECT*

Au lieu de fournir directement l'objet qu'il veut protéger, le fournisseur de la ressource encapsule cet objet dans un objet gardé et ne fournit que l'objet gardé à l'utilisateur.

Pour accéder à la ressource, l'utilisateur doit appeler la méthode *getObject* de l'objet gardé. Cette méthode consulte d'abord l'objet *Guard*, et la référence à l'objet n'est renvoyée que si l'objet *Guard* l'autorise.

Il est important de noter que **la classe *Permission* implémente l'interface *Guard***, ce qui permet d'utiliser n'importe quel objet dérivé de la classe *Permission* comme objet Guard sans programmation supplémentaire (voir l'exemple d'utilisation page suivante).

Les signatures des *GuardedObjects* sont les suivantes :

- `Public GuardedObject(Serializable objectToBeGuarded, Guard guard)`, le constructeur de la classe.
- `Public final Object getObject()`, permet de récupérer l'objet original à condition d'y être autorisé.

Exemple d'utilisation :

- Création de l'objet (côté fournisseur). La classe *FilePermission* est dérivée de *Permission* et implémente l'interface *Guard*.

```
FileInputStream f = new FileInputStream("/a/b/c.txt");
FilePermission p = new FilePermission("/a/b/c.txt","read");
GuardedObject go = new GuardedObject(f, p);
```

- Utilisation de l'objet dans un autre thread (côté utilisateur)

```
FileInputStream fis = (FileInputStream) go.getObject();
```

Une exception sera déclenchée si l'accès n'est pas autorisé pour ce thread.

4. ARCHITECTURES DISTRIBUÉES

4.1. INTRODUCTION

Dans les architectures d'aujourd'hui, la capacité d'un système à s'intégrer à d'autres systèmes informatiques est aussi importante que la qualité des services offerts. L'évolution du système dans un environnement hétérogène devient dès lors une préoccupation essentielle : une application doit pouvoir évoluer (et se déployer) rapidement sans que cela remette en cause l'existant.

De façon générale, on peut schématiser une application quelconque en disant qu'elle contient des processus qui émettent des requêtes (appelons-les processus appelants, ou *Clients*) et d'autres processus qui fournissent une réponse à ces requêtes (appelons-les processus répondants, ou *Serveurs*). Dans un tel schéma simplificateur, mettons en évidence le cas particulier où les processus appelants et les processus répondants sont situés sur des machines différentes (dites *distantes*).

Au départ, la communication entre Client et Serveur est réglée par une technologie appelée RPC (*Remote Procedure Call*). Ensuite, le développement des technologies orientées-objets a permis l'émergence de nouvelles technologies de distribution des applications, qui mettent en œuvre les concepts d'héritage, d'encapsulation et de polymorphisme.

Par **application distribuée**, on entend une application logicielle destinée à être exécutée sur plusieurs machines reliées par un réseau, que celui-ci soit local (LAN) ou non (p.ex. Internet). Il s'agit par conséquent d'un problème touchant à la fois les domaines du système d'exploitation, de la gestion réseau, et des techniques de programmation.

Dans l'infrastructure la plus simple répondant à cette exigence de travail coopératif, on aura une machine Cliente qui émet les requêtes, et une machine Serveur qui lui répond. Cette infrastructure à deux niveaux (2-*tiers*) permet une centralisation des traitements et des données au niveau du serveur. Une telle architecture est appelée **Client-Serveur 2-tiers**.

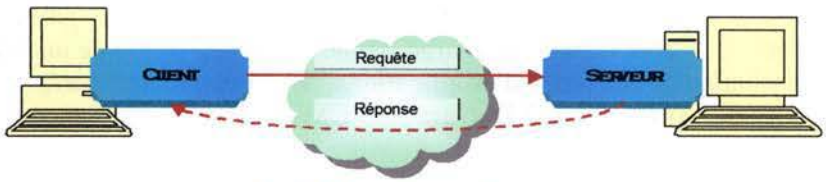


FIG. 37 – CLIENT-SERVEUR 2-TIERS

Une évolution de cette architecture de base propose de séparer les traitements de la gestion des données proprement dite : on obtient alors une architecture dans laquelle un Client s'adresse à un Serveur de Traitement, lequel est en relation avec un Serveur de Données. On parle alors assez logiquement de **Client-Serveur 3-tiers**.

Dans une perspective de meilleure répartition des services, la multiplication des serveurs intermédiaires entre le Client et le Serveur de Données offre la possibilité d'architectures **N-tiers**.

FIG. 38 – CLIENT-SERVEUR 3-TIERS

Il faut noter que ce type d'architecture n'impose pas forcément une répartition sur des machines différentes ; il s'agit avant tout d'une démarche de conception en couches applicatives indépendantes dont le dialogue est régit par des interfaces de communication particulières. Une telle conception vise une meilleure modularité et réutilisabilité du code. Une architecture distribuée est donc une architecture N-tiers qui fait abstraction de la localisation du code.

La distribution d'application se fait en utilisant un *Framework* qui joue le rôle de pont de communication entre les différents modules distribués en fournissant les outils nécessaires. Leur rôle central au cœur de l'organisation des modules justifie leur nom de **Middleware**. Par extension, la partie intermédiaire entre le Client et le Serveur dans une architecture 3-tiers ou N-tiers est également appelée *Middleware* ou *Middle-tier*.

Actuellement, on distingue 3 types ou normes d'architectures distribuées :
- DNA/.NET de Microsoft, avec le modèle DCOM.
- J2EE (Java) de Sun, avec le modèle RMI.
- CORBA de l'OMG (*Object Management Group*).

A ces normes viennent se greffer de nombreux services :
- la gestion des accès aux bases de données (ODBC, JDBC, OLE-OB) ;
- la gestion transactionnelle (OTS, JTS, JTA) ;
- la gestion des messages entre applications (MSMQ, JMS, MQ-Series) ;
- la gestion des événements (COM+ Event Service, CosEvent, CosNotification) ;
- l'interopérabilité entre les systèmes (DCE-RPC, JRMP, IIOP, RMI-IIOP, XML, SOAP) ;
- la sécurité ; etc.

Enfin, pour répondre notamment à des exigences techniques particulières telles que résistance aux pannes ou encore montée en charge (volumétrie), une classe de middlewares adaptés est apparue sous l'appellation de **Serveurs d'Applications**. Les 3 classes qui correspondent aux 3 types d'architecture sont COM+ (Microsoft), EJB (Java) et CCM (CORBA).

4.2. RPC (REMOTE PROCEDURE CALL)

[COUTURE 00] ◆ [DONSEZ 00]

RPC est un système essentiellement procédural, orienté sans connexion, qui « masque » l'appel à une procédure comme si celui-ci était local. L'application distribuée est divisée en un module client et un module serveur : le client appelle une procédure sur le serveur, le serveur exécute cette procédure et renvoie le résultat.

Les applications RPC doivent s'enregistrer auprès d'un *Port Mapper* pour être visibles pour les clients. Ceux-ci demandent ensuite au *Port Mapper* l'accès à un service particulier. Toute la

mécanique de transaction et de communication entre les deux parties distantes est gérée dans le *Stub* client et le *Skeleton* serveur.

Une couche de présentation XDR (*eXchange Data Representation*) permet d'uniformiser la présentation des données. RPC constitue un premier niveau d'abstraction important par rapport aux sockets ; ce système est à l'origine des autres modèles d'architectures distribuées.

4.3. DCOM (DISTRIBUTED COMPONENT OBJECT MODEL)

[COUTURE 00] • [WEBATRIUM 00]

Depuis 1993, l'architecture COM (*Component Object Model*) est l'architecture Microsoft de composants objets réutilisables ; elle permet en outre de s'affranchir du langage de programmation, un composant écrit dans un langage (p.ex.C++) pouvant être utilisé par un autre langage (p.ex.VB).

La version distribuée de ces composants objets s'appelle **DCOM** (*Distributed COM*, 1996), qui se base sur les couches DCE-RPC pour les appels distants. Depuis Windows 2000, l'appellation change en COM+, en intégrant quelques fonctionnalités supplémentaires (gestion d'événements, messages asynchrones, répartition de charge, etc.).

Etroitement liées au monde Microsoft, ces « normes » sont de facto réservées aux systèmes d'exploitation de la famille Windows 32 bits (95, 98, W2K, NT...), à laquelle elle est par ailleurs intimement intégrée notamment avec les contrôles ActiveX. Cette grande intégration des objets COM dans l'environnement Windows est donc à la fois la force de ce modèle (dans cet environnement) et sa faiblesse (en dehors de cet environnement). Bien que portable sous différents OS (il existe par exemple une mouture Unix de COM), cette architecture ne peut pas être qualifiée d'ouverte.

DNA (*Distributed iNternet Application*), l'architecture Internet de Microsoft, est bâtie autour de DCOM et regroupe plusieurs composants :

- le serveur d'application MTS (*Microsoft Transaction Server*),
- le middleware orienté message (*MOM*) MSMQ (*Microsoft Message Queue*),
- le serveur Web IIS
- le serveur de pages dynamiques ASP
- l'accès aux données OLE-DB.

Enfin, plus récemment, Microsoft a développé une architecture baptisée « **.NET** » (« *dot-net* »), qui n'est plus directement liée à COM+ (bien qu'en principe compatible). Destinée à intégrer tout le processus de création d'applications Web, « .NET » a été conçu dans l'optique d'un portage sous différentes plateformes, grâce au CLR (*Common Language Runtime*), une idée inspirée de la *Virtual Machine* de Java. Cette architecture orientée *WebServices* repose sur les technologies SOAP (*Simple Object Access Protocol*, que l'on peut rapprocher des Beans de Java), et SDL (*Service Description Language*), qui permettent d'appeler des objets via XML.

Notons que la plupart des éditeurs d'ORB CORBA (voir section suivante) proposent des ponts avec la technologie COM de Microsoft. COM peut également interagir avec CORBA en générant une interface COM autour d'un composant JavaBean, qui sert d'intermédiaire d'accès à CORBA grâce à Java (solution *bridge ActiveX/JavaBean* de Sun).

4.4. CORBA (COMMON OBJECT REQUEST BROKER ADAPTER)

[COUTURE 00] • [GEIB 98]

CORBA est une norme de distribution d'objets définie par l'OMG (*Object Management Group*), qui permet à des objets développés dans différents langages d'interagir au travers d'un réseau, grâce à un *mapping*, prévu actuellement pour les langages C, C++, ADA, Smalltalk, Java, Cobol et Objective C. L'architecture de Corba est *ouverte* ; elle permet l'intégration de systèmes hétérogènes.

ORB (*Object Request Broker*) est le *middleware* CORBA qui permet de détenir une référence virtuelle sur un objet distant ; il permet au client d'utiliser un objet CORBA situé sur le serveur distant comme s'il s'agissait d'un objet local, sans se soucier de sa localisation. Les objets de l'application distribuée ne communiquent donc pas entre eux ; ils passent toujours par un courtier d'objets. Les ORB communiquent eux-mêmes entre eux selon les spécifications de communication de l'**IIOP** (*Internet Inter-ORB Protocol*), protocole de communication inter-ORB sur Internet.

Les objets que l'on veut rendre accessibles au travers du réseau sont définis dans des interfaces publiques décrites au moyen du langage **IDL** (*Interface Definition Language*). Une telle interface décrit les services (signatures des méthodes et types de données concernés) offerts par l'objet CORBA, indépendamment de l'implémentation de l'objet.

Les interfaces IDL sont alors compilées pour un langage particulier, ce qui génère un proxy client (*stub*) et un proxy serveur (*skeleton*), qui jouent le rôle d'intermédiaires dans le dialogue entre objets client et objets serveur. En collaboration avec l'ORB, le stub et le skeleton assurent la transmission des paramètres et des valeurs de retour (via des opérations de *marshalling* et *unmarshalling*, qui permettent de traduire les arguments en flux binaires pour être envoyés à travers le réseau) au moyen du protocole IIOP. Ainsi, les *proxies* générés constituent l'interface publique qui encapsule l'objet distant, dont la logique applicative reste étrangère à CORBA.

L'architecture globale de construction d'applications réparties est décrite par l'OMG sous le nom d'*Object Management Architecture* (ou **OMA**), qui distingue les différents types d'objets intervenant dans une application distribuée, en fonction de leur rôle :

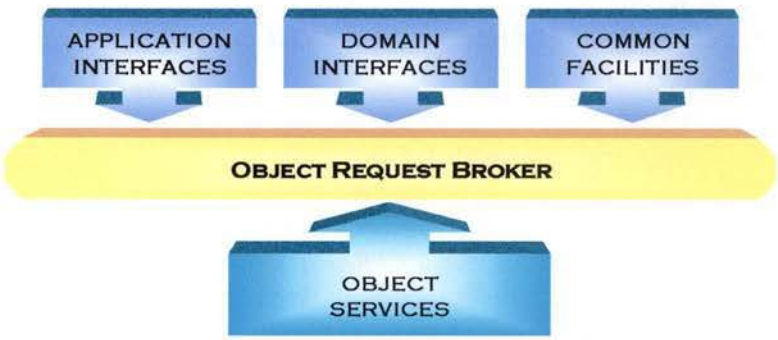


FIG. 39 – CORBA : OBJECT MANAGEMENT ARCHITECTURE

- L'ORB, aussi appelé *bus d'objets répartis*, constitue la clé de voûte de l'architecture CORBA ; il assure le transport des requêtes entre les objets CORBA, en masquant l'hétérogénéité des implémentations en termes de langages de programmation, de systèmes d'exploitation, d'architecture matérielle, et de technologies réseaux. En réalité, CORBA implémente uniquement les fonctionnalités de l'ORB, et non l'ensemble du *framework* OMA. Les services (appelés aussi *facilités*) disponibles pour le programmeur CORBA sont spécifiées autour de l'ORB.
- Les *services d'objets communs* (***Object Services***, ou CORBAServices), fournissent les services systèmes généralement nécessaires aux applications réparties, sous forme d'objets CORBA décrits au travers d'une interface IDL : *naming service* et *trading service* (le premier permet au client de trouver un objet à partir de son nom, le second à partir de ses propriétés), cycles de vie des objets, relations entre objets, transactions, concurrence, sécurité, etc.

- Les *utilitaires communs* (***Common Facilities***, ou CORBAFacilities), placés à un niveau d'abstraction supérieur, regroupent des *frameworks* (« canevas » d'objets) plus proches de l'utilisateur, standardisant par exemple l'interface utilisateur (p.ex. OpenDoc), le gestionnaire d'impression, la gestion de l'information, l'administration, le workflow, etc.
- Les *interfaces de domaines* (***Domain Interfaces***) proposent des objets-métiers spécifiques à des champs d'applications ou secteurs d'activités particuliers (finance, santé, télécommunications…). Les *Business Object Frameworks* (BOF) sont destinés à faciliter l'interopérabilité sémantique des applications développées pour un secteur d'activité commun (un même « métier »).
- Les *objets applicatifs* (***Application Interfaces***, ou Application Objects) regroupent les objets spécifiques à une application répartie particulière ; ils ne sont donc pas standardisés. Il s'agit d'objets spécialisés qui utilisent tous les services, utilitaires et interfaces.

LE BUS D'OBJETS RÉPARTIS Corba

Le bus CORBA est donc le « négociateur » à travers lequel les objets vont pouvoir dialoguer, en requérant des services auprès d'autres objets, sans rien devoir connaître de la localisation ou de l'implémentation de leur fournisseur de services (en dehors de leur interface publique).

L'ORB est divisé en plusieurs fonctionnalités distinctes :

L'*Object Request Broker* lui-même ; c'est le coeur de l'architecture responsable de la relation et de la communication entre les objets. Il permet la transparence du dialogue entre le client et le serveur, en prenant également en charge la gestion et la maintenance des échanges (gestion des erreurs, etc.). L'ORB comprend au moins les protocoles GIOP (*General Inter-ORB Protocol*) et IIOP (qui est une spécialisation de GIOP pour les réseaux TCP/IP).

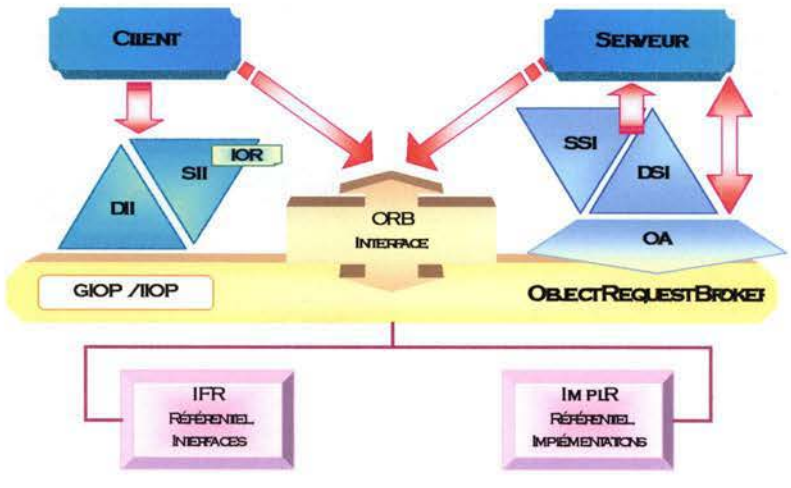


FIG. 40 – CORBA : BUS D'OBJETS RÉPARTIS

Le *Client* est celui qui désire invoquer les services d'un objet distant ; pour cela, il doit obtenir une référence de l'objet distant, afin d'accéder à sa structure logique (son interface) et ainsi invoquer ses méthodes.

Le *Serveur* est l'implémentation de l'objet distribué, qui ne publie que la sémantique de l'objet (les méthodes et les attributs rendus publics). Soulignons qu'un objet serveur pour un objet donné peut à son tour devenir client d'un autre objet (et vice versa).

La référence à un objet distant (on parle d'*Interoperable Object Reference*, ou **IOR**) est obtenue par le Client lors de l'étape de *binding* (connexion au serveur), et stockée dans une variable interne au *Stub* pour être ensuite utilisée à chaque accès à cet objet.

L'*Object Adapter* (ou **OA**) est l'outil (côté serveur) qui permet l'interaction entre les objets et l'ORB : c'est lui qui crée les objets CORBA, gérant leur activation automatique si nécessaire, et qui permet l'appel des méthodes de l'objet en lui transmettant les requêtes provenant du client.

L'IDL (*Interface Definition Language*) permet de rendre publique l'interface des objets distribués (opérations disponibles et technique d'invocation pour les clients). L'OMG a définit les règles de transformation (*mapping*) d'une interface IDL pour certains langages de programmation (déjà cités).

Le *Stub* (côté client) ou **SII** (*Static Invocation Interface*) et le *Skeleton* (côté serveur) ou **SSI** (*Skeleton Static Interface*) sont générés à la compilation de l'interface OMG-IDL ; ils servent d'interface (*proxy*) de communication entre le client et le serveur.

Un objet distant peut aussi être invoqué sans passer par le couple Stub/Skeleton, avec l'invocation dynamique d'objets distants : *Dynamic Invocation Interface* (ou **DII**) côté client, *Dynamic Skeleton Interface* (ou **DSI**) côté serveur. L'invocation dynamique permet de générer dynamiquement des requêtes vers tout objet CORBA dont on ne possède pas d'interface statique (Stub ou Skeleton).

L'*Interface Repository* (ou **IFR**) est un référentiel contenant une représentation des interfaces des objets distants utilisés lors de l'exécution de l'application. L'*Implementation Repository* (ou **ImplR**) contient les informations permettant à l'ORB de localiser et d'activer les différentes implémentations des objets référencés dans l'IFR. Ce référentiel est spécifique à chaque produit CORBA.

4.5. RMI (REMOTE METHOD INVOCATION)

[COUTURE 00] • [GEIB 98] • [HORSTMANN 00] • [WEBATRIUM 00]

RMI est la technologie d'invocation distante mise en œuvre par Sun ; la finalité est ici aussi de pouvoir appeler des méthodes d'un objet distant auquel on aurait accès. Les concepts de RMI sont souvent comparables à ceux de CORBA. Toutefois RMI est un système multiplateforme *mono-langage* (uniquement Java). Dans ce contexte, les objets et les données Java ont la même représentation d'une machine virtuelle à une autre, donc indépendamment de la plateforme. La connexion, la sélection du service approprié, la gestion des paramètres et de la valeur de retour sont pris en charge de façon transparente par RMI, via les proxies stub/skeleton.

L'application Java qui fait l'appel à une méthode distante est le client et celui qui traite cet appel est le serveur. Soulignons à nouveau que les rôles de client et de serveur sont dévolus *pour le temps d'un appel* ; lors d'un appel ultérieur le 'serveur' peut à son tour se trouver en position de 'client' (demandeur), et le 'client' en position de 'serveur' (fournisseur du service).

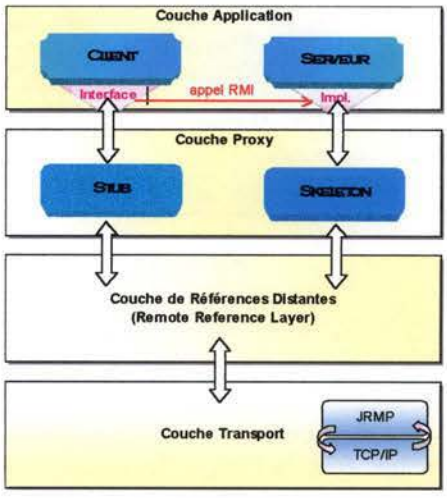


FIG. 41 – ARCHITECTURE RMI

La communication RMI peut être vue en 4 couches logiques : application, proxy, RRL et transport.

La couche application

Cette couche représente le client (demandeur de service) et le serveur (fournisseur du service). Les méthodes exportées par un serveur sont décrites dans une interface de définition de services (interface au sens Java, héritée de *java.rmi.Remote*) ; cette interface est implémentée dans une classe du serveur.

La couche *Proxy*

La couche Proxy comporte la partie *Stub* du client et la partie *Skeleton* du serveur, qui sont générées lors de la compilation de la couche application (via l'utilitaire *rmic*). Le rôle du *Stub* est d'intercepter

les appels clients vers les méthodes distantes, et de les rediriger vers le service RMI approprié. Le client doit préalablement localiser et référencer l'objet distant (voir processus de *Naming* ci-dessous).

Le Stub qui intercepte l'appel à une méthode d'un objet distant effectue une opération de *marshalling*, qui constitue en une sérialisation de tout objet en un stream. Côté serveur, le skeleton est chargé de l'opération inverse de *unmarshalling*, afin de procéder à l'appel de la méthode avec ses paramètres. Si la méthode produit une valeur de résultat, celle-ci suit le même procédé dans l'autre sens.

NOTE : Depuis Java 2, le *skeleton* n'est plus distingué du serveur (il lui est assimilé) et n'est donc plus généré par le compilateur *rmic* (sauf paramètre de compilation explicite indiquant la version 1.1).

La couche de références distantes (RRL : *Remote Reference Layer*)

Cette couche est chargée de gérer les références aux objets distants : RRL est une couche abstraite entre le transport réseau proprement dit et le Stub, qui permet à ce dernier de considérer qu'il dispose d'un protocole de type connecté (*Stream*), alors que le transport peut très bien être effectué via un protocole sans connexion (comme UDP).

Dans les premières versions de RMI, la connexion est de type Unicast, un client étant connecté à un serveur dont le service requis doit être préalablement instancié et exporté au système RMI. Depuis Java 2, RRL intègre ROA (*Remote Object Activation*) et gère ainsi les objets distants « dormants » : lors de l'appel à une méthode d'un objet distant, RMI vérifie l'état du service requis, et est capable de l'instancier à la demande. Java 2 permet également à un Stub de procéder à une requête Multicast en invoquant les services de plusieurs implémentations distantes d'un objet (et en acceptant la première réponse qu'il reçoit).

La couche transport (*Transport Layer*)

La couche Transport est chargée de gérer la connexion entre les machines virtuelles (les protocoles TCP et UDP peuvent être utilisés). Les connexions sont basées sur une adresse IP (ou un nom DNS) et des numéros de ports réseaux. RMI adjoint à TCP/IP un protocole propriétaire **JRMP** (*Java Remote Method Protocol*) ; notons que ce dernier oblige l'utilisation d'un skeleton côté serveur jusqu'à la version JDK1.1, et le rend obsolète à partie de JDK1.2. L'utilisation du protocole TCP/IP permet l'utilisation de connexions RMI locales (sur la même machine) ou réseau.

Enfin, il faut souligner que la conception en couches permet de remplacer la couche transport par une alternative. Ainsi, par exemple, l'indépendance des couches autorise l'utilisation du protocole ouvert IIOP de CORBA à la place du protocole JRMP spécifique à Java (voir *4.7. Extensions de la technologie RMI*, section *RMI-IIOP*).

Le service *Naming*

Le processus d'identification des objets distants sur le réseau est appelé *Naming*, par lequel un client peut obtenir la localisation d'un objet distant, se connecter au serveur concerné, et enfin invoquer les méthodes de l'objet distant.

RMI utilise un gestionnaire de registres (**rmiregistry**), qui joue le rôle d'annuaire des services dans lequel le serveur s'enregistre et par lequel le client peut retrouver l'adresse de l'objet recherché. L'idée est d'associer un objet serveur avec un nom public identifiable par les clients. Toute machine serveur d'objets distants doit disposer d'un registry (actif par défaut sur le port 1099).

En utilisant une adresse URL (*Uniform Resource Locator*), un client accède au registry du serveur par une opération de *lookup*, qui permet de localiser l'objet recherché. Le lookup du client accède au

rmiregistry du serveur en passant par le Stub, et obtient en retour une référence distante à l'objet (contenant notamment le nom du serveur et le numéro de port à utiliser pour se connecter à la machine virtuelle proposant l'objet distant). La couche transport utilise ces informations pour ouvrir une connexion de type socket avec le serveur, et la référence est transférée à la couche application.

4.6. COMPARAISON ET CHOIX DES TECHNOLOGIES

[COUTURE 00] • [DONSEZ 00] • [HORSTMANN 00]

Nous ne nous étendrons pas davantage sur les RPC, technologie limitée dont l'intérêt est ici purement historique (RPC est un concept procédural, sans gestion d'objet, sans référence distante, et dépendant de la localisation du serveur).

En ce qui concerne les technologies RMI, CORBA et DCOM, on peut établir certains critères de comparaison entre eux, qui peuvent aider à faire un choix lors d'un projet particulier. Quelques uns de ces critères sont repris dans le tableau ci-dessous (partiellement d'après [COUTURE 08-00]). Chaque critère est apprécié selon trois niveaux d'adéquation selon la technologie :

FIG. 42 – TABLEAU : CRITÈRES DE COMPARAISON DES TECHNOLOGIES RMI, CORBA ET DCOM

▫	= Non	La technologie visée ne remplit pas ce critère.
✓	= Oui	La technologie remplit ce critère.
☑	= Absolument	La technologie remplit particulièrement ce critère.

Critère	RMI	CORBA	DCOM	Commentaire
Standardisation	▫	☑	▫	Standard de l'OMG
Rapidité d'exécution	▫	✓	✓	Java n'est pas compilé
Rapidité d'apprentissage	☑	✓	▫	RMI plus simple à maîtriser, puis CORBA
Rapidité de développement	☑	✓	▫	RMI plus simple à maîtriser, puis CORBA
Multi-langages	▫	☑	✓	RMI=Java; DCOM=Microsoft C++,VB,J++
Multi-plateformes	☑	☑	▫	DCOM technologie Windows
Technologie O.O.	☑	☑	✓	DCOM suivant les versions
Evolutivité	✓	☑	✓	Avantage de Corba comme standard
Opérations distantes	☑	✓	✓	Opérations plus complètes RMI
Langage d'interface	▫	☑	✓	Corba IDL ; Microsoft IDL propre
Simplicité	☑	✓	▫	
Support des applets	☑	☑	✓	DCOM limité à J++
Tolérance de pannes	▫	☑	▫	
Répartition des charges	▫	☑	▫	
Annuaire d'objets	☑	☑	▫	RMI Registry ; CORBA Naming
Appels statiques/dynamiques	✓	✓	✓	RMI depuis Java 2
Activation d'objets auto.	✓	✓	✓	
Appels asynchrones	✓	☑	▫	

A ces critères « mesurables », il faut encore ajouter certains critères très importants qui influencent le choix d'une technologie indépendamment des qualités de la technologie elle-même ; notamment :

- L'expertise des membres de l'équipe-projet ;
- Le type d'application à développer (pour lequel une passerelle n'est pas toujours adéquate) ;
- Le *legacy system* dans lequel s'inscrit le développement (technologies, langages, plateforme…).

Etant donné notre objectif-phare qui consistait à étudier les technologies liées à Java, et au vu de son caractère non-ouvert, l'environnement DCOM nous intéresse moins dans le cadre de ce travail.

Comme nous l'avons vu, CORBA est destiné à faire dialoguer des objets provenant d'environnement de programmation hétérogènes, alors que RMI est intimement lié à Java. C'est finalement là le critère de sélection essentiel entre ces technologies. Si les objets qui communiquent sont tous écrits en Java, l'universalité de CORBA n'est pas nécessaire. La solution à mettre en œuvre peut alors être envisagée avec moins de complexité et de lourdeur, grâce à RMI, une solution dédicacée Java plus simple à mettre en œuvre et à maîtriser.

Pour les raisons contraires à celles qui ont écarté DCOM de notre étude, la technologie RMI constitue donc pour nous un centre d'intérêt évident, et s'impose dès à présent comme choix principal dans la suite de notre travail. Quant à CORBA, son universalité et surtout les rapprochements effectués ces dernières années avec le monde Java, et en définitive la relative adéquation de ces deux technologies, peuvent éventuellement justifier quelques investigations techniques particulières (voir par exemple l'extension RMI-IIOP section suivante).

4.7. EXTENSIONS DE LA TECHNOLOGIE RMI

[COUTURE 00] • [ANDOH 99]

Avec l'utilisation des sockets, JRMP est confronté à un problème particulier lorsque la connexion RMI transite par des firewalls, dont le but premier est de bloquer tout trafic réseau sauf celui associé à certains ports clairement identifiés (comme le port 80 p.ex.).

Il faut distinguer deux cas de figure : celui où c'est le client qui se trouve derrière un firewall, et celui où le serveur se trouve aussi derrière un firewall. Dans le premier cas, le firewall empêche le transfert de l'appel à l'extérieur du firewall, dans le second cas il empêche la réception de l'appel.

Pour contourner ce type de problème où un firewall empêche la connexion avec un serveur RMI, la couche de transport de RMI permet d'encapsuler les appels JRMP dans des requêtes de type HTTP POST (pour sortir via un firewall), ou dans des paquets HTTP transférés via le port 80 du serveur (qui doivent être décodés dès la réception sur le serveur par la machine virtuelle Java, pour être retransférés sur le bon port). Il est à noter que ces manipulations entrainent évidemment une perte de performance.

Parmi les atouts sécurité de RMI, il faut également souligner que la conception en couches de cette technologie permet de remplacer la couche transport par une alternative. Dans le cadre qui nous occupe, cela implique qu'il est possible de modifier la couche Transport pour permettre la transmission de streams chiffrés, ou l'intégration d'autres algorithmes de sécurité (digests, signatures, etc.), et cela sans aucune influence logique sur les couches supérieures.

RMI - IIOP

La conception en couches, avec la possibilité de remplacer la couche transport, a permis d'envisager un rapprochement de RMI avec la technologie CORBA : RMI peut ainsi fonctionner avec le protocole ouvert IIOP à la place du protocole propriétaire JRMP.

Historiquement, les technologies RMI et CORBA n'ont pas été développées dans l'optique de pouvoir communiquer ensemble. JRMP notamment est incapable d'interagir avec d'autres protocoles, ce qui pose de sérieuses difficultés lorsqu'un nouveau développement doit pouvoir utiliser les ressources existantes d'un *legacy system* hétérogène, développé avec des technologies non Java. C'est dans ce contexte que le modèle CORBA devient indispensable.

Le rapprochement des deux technologies, permettant de bénéficier de la facilité de mise en œuvre de Java-RMI et l'interopérabilité de CORBA, est donc venu avec une interface hybride RMI-IIOP, qui supporte en réalité à la fois JRMP *et* IIOP. Une classe supportant cette interface peut alors être exportée simultanément en version JRMP et en version IIOP, sans avoir à réécrire le code Java (seuls des paramètres d'exécution sont à définir). Cette opération est appelée *dual export*, et est réalisée via le compilateur *rmic* (additionné des paramètres adéquats *–iiop* pour la génération du stub et du skeleton, et *–idl* pour la génération du fichier IDL correspondant à l'interface RMI).

Une interface IDL est donc générée pour l'objet RMI, ce qui permet à un client de l'utiliser comme s'il était un objet CORBA. Alors que nativement les univers RMI et CORBA sont hermétiques l'un vis à vis de l'autre, RMI-IIOP offre de grandes possibilités d'interaction des deux mondes.

FIG. 43 – TABLEAU : INTEROPÉRABILITÉ RMI-CORBA AVEC IIOP

Serveur / Client	RMI (JRMP)	RMI (IIOP)	CORBA
RMI (JRMP)	☑	☑	▫
RMI (IIOP)	☑	☑	✓
CORBA	▫	☑	☑

Comme le montre le tableau ci-contre, les possibilités d'interopérabilité les plus complètes sont obtenues avec un serveur RMI-IIOP, qui peut interagir avec tout type de client (RMI, RMI-IIOP, CORBA).

Un client RMI-IIOP peut également interagir avec tout type de serveur, sauf dans le cas d'un serveur purement CORBA, avec lequel les interactions sont restreintes pour certains types d'opérations. Cette restriction est due au fait que RMI-IIOP n'implémente pas la totalité de la sémantique CORBA, mais seulement le sous-ensemble de cette sémantique utile à une interface avec des objets RMI. Les services CORBA non reconnus par RMI-IIOP ne sont donc pas accessibles.

Les deux grandes adaptations de CORBA aux spécifications RMI concernent la transmission des objets par valeur et le *mapping* Java-IDL. La transmission des objets par valeur, nativement supportée par RMI grâce à la sérialisation des objets Java, a été introduite dans CORBA pour offrir aux autres langages une fonctionnalité similaire. Le *mapping* Java vers IDL permet quant à lui de convertir les interfaces Java RMI dans le langage de définition standard CORBA IDL. Ces adaptations sont prises en compte à partie de la version 2.3 de CORBA, et de la version 1.3 du JDK de Sun.

Notons encore que certaines spécifications RMI nécessitent quelques adaptations pour la compatibilité avec IIOP. Pour information, citons par exemple le mécanisme de *garbagge collection* distribué , qui n'est pas supporté par CORBA ; le mécanisme de registry de RMI qui est remplacé par le *naming* de JNDI ; ou l'activation d'objets RMI qui est remplacée par le *portable object adapter* de CORBA. L'investigation de ces particularités techniques dépasse toutefois le cadre de ce mémoire.

CONCLUSION PARTIE I

La première partie de ce mémoire nous a montré que Java est un langage de développement orienté dès sa conception vers la sécurité. Cette orientation est perceptible depuis le degré de visibilité que l'on peut attribuer aux classes, méthodes et variables d'instance, jusqu'à la gestion très fine de l'accès aux ressources par le Gestionnaire d'Accès, en passant par les contrôles d'intégrité du compilateur, le vérificateur de bytecode, le chargeur de classe personnalisable (avec le mécanisme de délégation), et le Gestionnaire de Sécurité dans les APIs.

Nous avons vu également que la plateforme fournit une architecture de sécurité regroupant :
- la gestion des condensés de message et des signatures numériques, la gestion des certificats, et la gestion des clés, ainsi que la mise en place du concept de CSP (**JCA**) ;
- l'implémentation des services cryptographiques de chiffrement et d'échange de clés, ainsi qu'un mécanisme original de gestion d'objets signés/scellés/gardés (**JCE**) ;
- l'identification et l'authentification, ainsi que la connexion à un domaine de sécurité (**JAAS**) ;
- la gestion d'une communication sécurisée via SSL (**JSSE**).

Ces mécanismes, parfois originaux, permettent de rencontrer la plupart des services de sécurité décrits dans le premier chapitre (*Sécurité de l'Information*).

Ainsi, JCE et JSSE proposent l'infrastructure nécessaire pour implémenter des services tels que la **confidentialité** (chiffrement) ou l'**authenticité** (contrôle d'intégrité) des données. Les outils utiles à la **vérification de l'identité** sont fournis par JCA (gestion des certificats) et sont complétés par JAAS (services avancés d'authentification). Les **permissions d'accès** sont régies nativement par les mécanismes de Gestionnaire de Sécurité et de Contrôleur d'Accès, appuyés par l'infrastructure JAAS pour une gestion améliorée des autorisations.

Toutefois, nous soulignerons le fait que d'autres services fondamentaux de sécurité manquent à l'infrastructure actuelle de Java : outre le défaut de gestion de la **disponibilité du service** ou d'une préoccupation récente comme l'**anonymisation**, on notera surtout l'*absence d'interface standard de* ***non-répudiation*** et d'***audit***.

L'implémentation d'une application sécurisée basée sur Java doit tenir compte des spécificités du langage, de ses nombreux avantages par rapport à ses concurrents, mais aussi de ses manquements ou de ses failles. Dans la seconde partie de ce mémoire, nous nous consacrons à la modélisation de la sécurité durant la conception d'une application Java. Nous reviendrons à cette occasion sur les types d'attaques possibles à l'encontre d'un système, avec les réponses Java appropriées.

PARTIE II :

CONCEPTION ET MODELISATION D'APPLICATIONS SECURISEES

Dans cette seconde partie, nous nous intéressons à la conception et à la modélisation d'applications sécurisées. Nous décrivons pour cela une méthode basée sur la notation standard UML.

Ensuite nous nous intéressons à l'extension de cette notation pour la modélisation de la sécurité, principalement à travers UMLsec.

Enfin, nous proposons un Guide pour la sécurisation d'un projet Java, en reprenant les bases de modélisation décrites ainsi que les techniques détaillées dans la première partie de ce mémoire.

5. MODÉLISATION

Dans cette seconde partie, nous allons chercher à développer un guide de mise en œuvre des concepts de sécurité que nous avons décrits dans la première partie.

5.1. MODÉLISATION UML

[FOWLER 01] • [FUNDP 00] • [LOPEZ 98]

UML (*Unified Modeling Language*) a succédé à une série de méthodes d'analyse et de conception orientées objet apparues à la fin des années quatre-vingts. Il unifie principalement les méthodes de Booch, Rumbauch et Jacobson. UML a été normalisé par l'OMG (*Object Management Group*) et fait maintenant partie de ses standards.

Grâce à cette normalisation par l'OMG, UML est devenu aujourd'hui le standard *de facto* du monde industriel pour la modélisation orientée-objet.

Il est important de préciser qu'UML n'est pas une méthode ou une démarche d'analyse mais un simple **langage de modélisation**, c'est-à-dire une notation essentiellement graphique utilisée pour décrire les différentes étapes d'analyse et de conception.

Le succès d'UML réside dans la simplicité et la clarté des différents schémas et diagrammes qu'il permet de produire. En effet, les étapes successives de l'élaboration d'un logiciel conduisent à de nombreux échanges entre les différents intervenants d'un projet (décideurs, utilisateurs, analystes, programmeurs) ; à chacune de ces étapes, il est possible d'associer un **modèle** qui peut être décrit par une notation UML.

« Le principal intérêt de l'utilisation d'UML se trouve dans les possibilités de **communication** qu'il offre. (...) Le langage naturel est trop imprécis et brouillon quand les concepts se complexifient. Le code est précis, mais trop détaillé. Nous utilisons donc UML lorsque nous avons besoin d'une certaine précision, sans toutefois nous perdre dans les détails. UML sert à souligner les détails qui sont importants. » [FOWLER 01]

Au-delà des techniques proposées par UML, il est nécessaire de suivre une **méthode de développement**. « Une méthode est en quelque sorte un guide définissant les règles de passage d'un modèle à un autre de façon à tendre progressivement vers le modèle final de l'édification du logiciel. »

« Un modèle offre une vision simplifiée du problème. Il constitue une représentation abstraite d'une réalité. Il fournit une image simplifiée du monde réel. En cela, un modèle est toujours incomplet, mais il doit être juste. Tous les éléments du modèle doivent être impliqués dans le domaine d'application. Par contre, tout élément du domaine d'application ne doit pas forcément se retrouver dans le modèle. Seuls les éléments intéressant la mise en œuvre du système doivent être représentés dans le modèle. » [LOPEZ 98]

Nous allons nous inspirer de la méthode classique, dite « en cascade ». Notre but n'est pas de décrire complètement cette méthode de développement mais simplement de fournir un cadre de travail dans lequel nous pourrons montrer où se situent les besoins en sécurité et quels sont les éléments à utiliser pour construire une application sécurisée. A chaque phase nous associerons les modèles UML qui nous ont semblé les plus pertinents.

Les différents concepts exposés dans cette section seront illustrés à travers une application exemple décrite dans la troisième partie.

La figure ci-dessous présente les différentes étapes de la méthode en cascade :

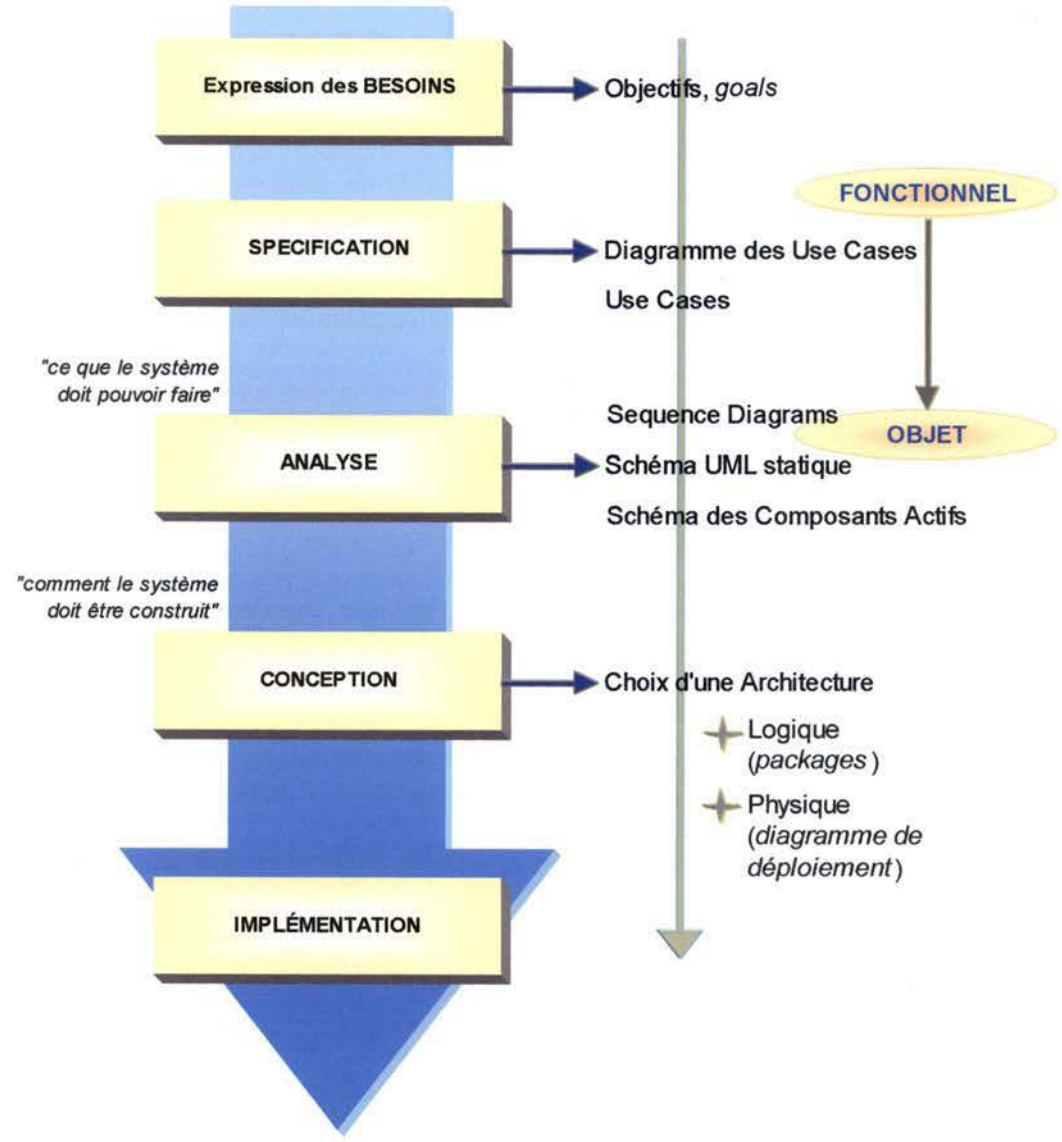


FIG. 44 – MÉTHODE DE DÉVELOPPEMENT EN CASCADE

5.1.1. EXPRESSION DES BESOINS

« L'**expression des besoins** doit traduire ce que le futur système est susceptible d'apporter aux utilisateurs, en faisant abstraction de la manière dont il sera construit. Cette première phase du développement se focalise donc sur les **propriétés externes** du logiciel.» [LOPEZ 98]

5.1.2. SPÉCIFICATION

La phase de **spécification** permet de préciser et compléter les besoins des utilisateurs. Il s'agit toujours d'une phase « orientée utilisateurs » qui doit être élaborée par des experts du domaine d'application.

La technique des « ***use cases*** » due à Jacobson est reprise dans le formalisme UML. Elle constitue une bonne approche pour spécifier le système. « Un cas d'utilisation est un ensemble de scénarios reliés par un objectif commun, celui de l'utilisateur. (...) On rencontre souvent des cas d'utilisation avec un cas commun, "tout se passe bien", et plusieurs cas alternatifs où les choses se passent mal, ou se passent bien, mais autrement. » [FOWLER 01]

Jacobson propose également un diagramme permettant de visualiser les use cases et d'obtenir une vue d'ensemble du système.

« Le **diagramme des use cases** permet de voir de façon simple :
- les différents acteurs ;
- comment est délimité le système ;
- les fonctionnalités demandées au système ;
- les rôles des différents acteurs vis-à-vis du système. » [LOPEZ 98]

En fait, la spécification UML définit un acteur comme un ensemble de **rôles** que l'utilisateur peut jouer par rapport au système. Un acteur a un rôle pour chaque use case avec lequel il communique. Un même utilisateur peut correspondre à plusieurs acteurs s'il assume plusieurs fonctions dans le système : par exemple un responsable commercial peut aussi remplir la fonction de commercial.

5.1.3. ANALYSE

Nous entrons ensuite dans la phase d'**analyse** : cette « phase d'analyse permet de s'accorder sur "ce que doit faire le système" avant de s'accorder sur la "manière dont il doit le faire". » [LOPEZ 98]

A partir des *use cases*, il est facile d'établir les **diagrammes d'interaction**. Il s'agit de « modèles qui décrivent la façon dont des groupes d'objets collaborent pour réaliser un comportement donné. Un diagramme d'interaction type capture le comportement d'un seul cas d'utilisation. Il représente un certain nombre d'objets et les messages qui sont transmis entre ces objets dans la réalisation du cas d'utilisation. » [FOWLER 01]

Le type de diagramme d'interaction que nous utiliserons par la suite est le **diagramme de séquence** : « un objet y est représenté sous la forme d'un rectangle au sommet d'une ligne pointillée verticale, la ligne de vie de l'objet. (...) On représente chaque message par une flèche entre les lignes de vie de deux objets. L'ordre dans lequel ces messages sont représentés est de haut en bas. » [FOWLER 01]

Nous avons opté ici pour une convention graphique particulière : une flèche pleine représente un appel synchrone pour lequel il existe un retour représenté en pointillé tandis qu'une flèche creuse représente un appel asynchrone. Il est aussi possible de représenter un auto-appel (c'est-à-dire un message que l'objet s'envoie lui-même) par une flèche repointant sur l'objet de départ.

Nous tenons à mettre en évidence la construction du *diagramme de séquence* à partir du *use case*. Il s'agit ici d'un des pivots de la méthode que nous utilisons : il permet de passer aisément de la représentation de l'utilisateur, la plus naturelle, essentiellement **orientée fonction**, vers une représentation **orientée objet**. Il s'agit d'un changement de cap important : à partir de cette étape, nous construirons notre système autour de la notion d'objet plutôt qu'autour de la notion de fonctionnalité.

Nous avons eu l'occasion d'expérimenter l'efficacité de cette méthode au cours de laboratoire de développement de logiciels suivi durant l'année académique 2000-2001.

Le *diagramme de séquence* dérivé d'un *use case* n'est évidemment qu'une vue fragmentaire de notre système. Pour en obtenir une vue **globale**, il est nécessaire d'établir le *diagramme de classe* correspondant. Les *diagrammes de classe* font partie intégrante d'UML et constituent l'épine dorsale de la plupart des méthodes orientées objet.

« Un diagramme de classe décrit les types d'objets qui composent le système et les différents types de relations statiques qui existent entre eux. Il existe principalement deux sortes de relations statiques : les **associations** et les **sous-types**. Les diagrammes de classe représentent également les **attributs** et les **opérations** d'une classe et les contraintes qui s'appliquent à la façon dont les objets sont connectés. » [FOWLER 01]

Le diagramme de classe offre une vue d'ensemble sur le système qui permet de l'utiliser selon différents points de vue :

- **Point de vue conceptuel** : chaque classe représente un concept du domaine étudié ;
- **Point de vue des spécifications** : on s'intéresse aux différentes interfaces sans tenir compte de l'implémentation ;
- **Point de vue de l'implémentation** : ici nous avons réellement des classes au sens de la programmation objet.

« Comprendre de quel point de vue on se place est crucial, tant pour la création que pour l'interprétation des diagrammes de classe. Malheureusement, la délimitation entre eux est quelque peu floue, et la plupart des modélisateurs ne se préoccupent pas de dissiper la confusion. Si, selon nous, la différence entre le point de vue conceptuel et celui des spécifications est rarement significatif, il est très important de séparer le point de vue des spécifications de celui de l'implémentation. »
[FOWLER 01]

5.1.4. CONCEPTION

Dans la progression de notre méthode, la phase suivante est la phase de **conception** : elle « permet de s'accorder sur "la manière dont le système doit être construit" et non plus sur "ce qu'il doit faire". » [LOPEZ 98]

C'est au cours de cette phase que seront définies les **architectures logique et physique**. Nous revenons ici à une des plus anciennes questions méthodologiques en matière de logiciel : décomposer un système en sous-systèmes. Vu l'approche orientée-objet que nous avons adoptée, il ne s'agit évidemment pas d'une découpe fonctionnelle mais bien d'un regroupement d'objets.

Du point de vue de la conception logique, l'idée consiste à regrouper les classes en unités de plus haut niveau. En UML, ce mécanisme de regroupement se nomme un *package*. Le regroupement s'effectue sur base des dépendances entre les objets. « Il y a dépendance entre deux éléments si une modification apportée à l'un d'entre eux provoque un changement dans l'autre. » [FOWLER 01]

Du point de vue de la conception physique, un *composant* représente un module de code physique ; les classes y sont regroupées suivant la localisation physique du code. Un composant correspond souvent

à un package, sauf lorsqu'une classe d'un package doit se retrouver dans plusieurs localisations de code ; auquel cas la classe se trouve dans plusieurs composants bien qu'elle ne se trouve que dans un seul package logique.

UML propose un « *diagramme de déploiement* qui montre les relations physiques qui existent entre les composants matériels et logiciels du système. (...) Chaque *nœud* d'un diagramme de déploiement représente une unité informatique – dans la plupart des cas matérielle, du capteur au mainframe. (...) D'autre part, un diagramme de composants permet de représenter les différents composants existant au sein d'un système et leurs dépendances.» [FOWLER 01]

Souvent, ces deux diagrammes sont confondus, ce qui permet de montrer quels composants s'exécutent sur quels nœuds. Toutefois, lorsque l'architecture devient complexe, il peut être utile de construire d'abord un schéma de l'architecture logique du système indépendamment des localisations, afin de faciliter la réflexion lors de l'intégration des liaisons physiques.

5.1.5. IMPLÉMENTATION

« La phase d'**implémentation** est la phase au cours de laquelle les structures et les algorithmes définis pendant la conception sont traduits dans un langage de programmation et/ou une base de données. » [LOPEZ 98]

Dans le cadre de ce mémoire les phases de vérification, de validation et de maintenance, quoique d'une importance capitale dans un projet réel, ne seront pas abordées. En effet, notre but n'est pas de décrire de façon détaillée une méthode de construction de logiciel mais bien de montrer **l'intégration de certains aspects de la sécurité** dans la gestion d'un projet Java.

5.1.6. AUTRES DIAGRAMMES

En plus des diagrammes que nous avons déjà présentés, UML a repris d'autres diagrammes qui permettent d'apporter davantage de précision sur certains points particuliers d'un projet. Il s'agit des diagrammes d'états (ou diagrammes d'états-transitions - *Statechart diagram*) et les diagrammes d'activité (*Activity diagram*).

Le **diagramme d'états** permet de décrire tous les états possibles que peut prendre un objet au cours de sa vie. La transition, c'est-à-dire le passage d'un état à un autre, a lieu en fonction d'une condition et des évènements que l'objet reçoit selon la syntaxe *Evènement [Garde] / Action*. Le franchissement d'une transition a lieu lorsque l'événement s'est produit, que la condition (garde) est vérifiée et après que l'action soit terminée. (Note : pour éviter toute confusion avec la notion d'objet gardé, nous utiliserons dans ce contexte le terme de *condition* plutôt que *garde*).

A chaque état est associé une activité, notée *do/activité*. La différence entre action et activité est relativement subtile et tient à la rapidité d'exécution des processus : une action s'exécute « plus rapidement » qu'une activité. En outre, une activité peut être interrompue (par un événement), contrairement à une action.

L'exemple ci-contre représente le *diagramme d'états* relatif à la saisie-validation d'un

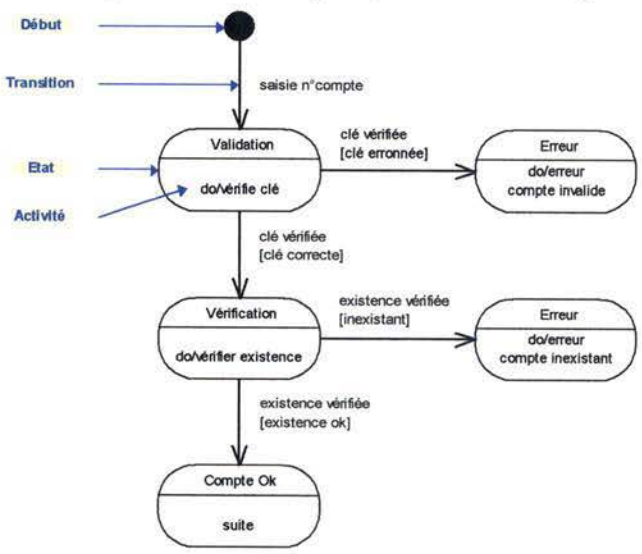


FIG. 45 – UML : DIAGRAMME D'ÉTATS (EXEMPLE)

numéro de compte en banque. Remarquons que la première transition ne comporte qu'une action qui s'exécute directement. La transition vers l'état « Vérification » est franchie dès que la saisie est terminée.

A partir d'un état donné, une et une seule transition peut se réaliser, ce qui impose des conditions mutuellement exclusives (clé erronée *vs* clé correcte).

« Les diagrammes d'états conviennent bien pour décrire le comportement d'un objet à travers plusieurs cas d'utilisation. Ils conviennent mal pour décrire un comportement qui implique plusieurs objets qui collaborent. C'est pourquoi il est utile de les combiner avec d'autres techniques. » [FOWLER 01]

Les **diagrammes d'activité** permettent la description des activités correspondant à un processus en montrant sur un même diagramme les comportements **conditionnels** et les comportements **parallèles**.

Le comportement conditionnel est introduit par un *branchement*, qui s'ouvre sur plusieurs choix mutuellement exclusifs, et s'achève sur une *fusion*. Le comportement parallèle permet de décrire plusieurs activités simultanées entre deux barres de synchronisation, appelées *débranchement* et *jonction*. Le diagramme peut être éventuellement complété par des *travées* qui permettent de spécifier les responsables (personnes ou classes) de chaque activité.

La figure ci-dessous représente le diagramme d'activité d'une prise de commande.

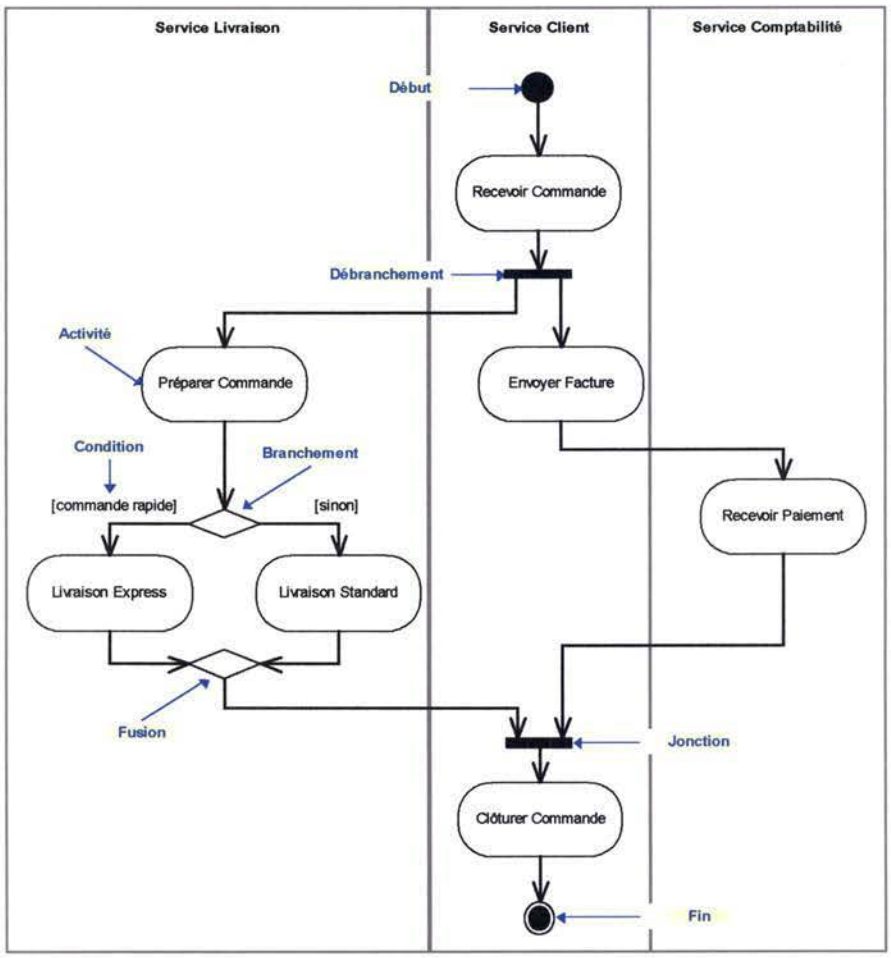


FIG. 46 – UML : DIAGRAMME D'ACTIVITÉ (EXEMPLE)

« La grande force des diagrammes d'activité réside dans le fait qu'ils supportent et qu'ils encouragent les comportements parallèles, ce qui en fait un excellent outil de modélisation des *workflows* et, en principe, de programmation multi-thread. Leur gros inconvénient est qu'ils ne font pas apparaître très clairement le lien existant entre actions et objets. » [FOWLER 01]

5.2. MODÉLISATION UMLSEC

[JURJENS 01] • [JURJENS 01B] • [JURJENS 02]

Une conclusion découle de notre étude sur la sécurité de l'information et notamment des différentes attaques que nous avons répertoriées : les logiciels sont à la base de la plupart des problèmes de sécurité informatique. Nous voulons dire par là que les bases de la cryptographie sont à présent bien maîtrisées, que leurs limites sont connues et que la plupart des pirates informatiques ne créent pas de brèches dans ces mécanismes de sécurité, ils se contentent d'exploiter les failles existant dans les programmes qui les utilisent.

Bien sûr, ces failles ne sont pas créées de façon intentionnelle. Il reste néanmoins assez étonnant que face aux conséquences souvent désastreuses des « trous » de sécurité, la construction des logiciels ne prenne pas davantage en compte les aspects de la sécurité.

Les méthodes gravitant autour de la notation UML constituent un réel progrès pour concevoir des logiciels de plus grande qualité, dépourvus d'erreurs et répondant mieux aux attentes des utilisateurs. N'est-il pas surprenant toutefois que la notation UML ne propose aucun modèle pour exprimer les différents mécanismes de protection à utiliser pour sécuriser un logiciel ?

Face à ce constat, nous avons été agréablement surpris par la découverte des travaux de Jan Jürjens, chercheur au département informatique de l'université de Munich (Allemagne) et de l'université d'Oxford (Royaume-Uni). Ce dernier propose des pistes pour une extension de la notation UML, baptisée **UMLsec**, ainsi qu'une formalisation de la notation UML dans le but d'automatiser la vérification des besoins en sécurité.

Nous nous sommes particulièrement intéressés à la première partie de ses travaux, d'autant qu'il utilise la plupart des diagrammes UML qui viennent d'être décrits pour traiter des aspects de la sécurité.

5.2.1. BUTS POURSUIVIS

Parmi les buts d'une extension de la notation UML pour le développement de systèmes sécurisés nous pouvons citer :

- l'évaluation des différents modèles UML à la recherche des vulnérabilités du système
- l'encapsulation des règles de « bonne pratique » de sécurité
- la disponibilité des mécanismes de sécurité pour les développeurs non spécialisés en sécurité
- la prise en considération de la sécurité dès le début d'un projet dans les différentes phases de modélisation en tenant compte du contexte où s'insère le système
- un meilleur rapport qualité/prix des coûts de développement d'un projet.

UML offre des capacités d'extension par l'ajout de légendes sur les diagrammes. Il existe deux possibilités : soit des **stéréotypes** (écrits entre guillemets doubles : « stéréotype ») ou des paires étiquette-valeur (écrites entre accolades : {étiquette, valeur}).

Dans le même esprit, l'ajout de **contraintes** sur les diagrammes permet de raffiner la sémantique des éléments stéréotypés. En effet, dans un diagramme de classe, par exemple, les constructions fondamentales – associations, attributs et généralisation – expriment les contraintes importantes, mais elles ne peuvent pas toutes les indiquer. UML permet d'ajouter d'autres contraintes par l'emploi de texte placé entre accolades.

Enfin, l'ensemble des extensions correspondant à un but particulier peut faire l'objet d'un **profil**. Par exemple, l'OMG a créé un profil IDL (*Interface Definition Langage*) CORBA.

Le profil UMLsec a pour base les principes suivants :

- la création de stéréotypes pour modéliser les exigences récurrentes en matière de sécurité (confidentialité, intégrité,...)
- l'utilisation de contraintes particulières pour évaluer les spécifications et mettre en évidence les vulnérabilités potentielles.

Pour chaque diagramme, nous allons examiner comment UMLsec utilise ces extensions simples, fournies en standard par UML pour exprimer les besoins en sécurité d'une application.

5.2.2. DIAGRAMME DE USE CASES

Le diagramme de use cases peut être utilisé pour montrer les besoins en sécurité. Par exemple, le diagramme de use cases de la figure suivante décrit une situation simple : un client achète une marchandise à un vendeur. Il est possible d'y ajouter le stéréotype « échange équitable » dont la sémantique intuitive implique que les deux actions 'acheter' et 'vendre' sont liées, en ce sens que si une des deux est exécutée, l'autre devra l'être aussi.

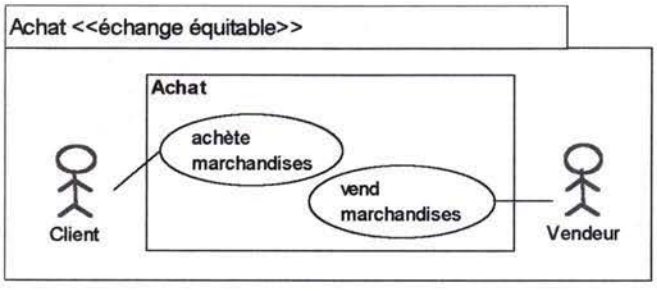


FIG. 47 – UMLSEC : DIAGRAMME DE USE CASES (EXEMPLE)

5.2.3. DIAGRAMME D'ACTIVITÉ

Comme nous venons de le voir, le diagramme d'activité est spécialement utile pour modéliser un *workflow* et pour expliquer un use case avec plus de détails.

Au départ de l'exemple précédent, il est possible de tracer les deux diagrammes d'activité représentés dans la figure suivante. Le premier ne remplit pas l'exigence « échange équitable » car le vendeur peut ne jamais livrer la commande. Dans le second, par contre le stéréotype « avec preuve » précise que l'acheteur doit avoir une preuve de paiement ce qui lui permettra d'introduire une réclamation si la marchandise n'a pas été livrée à la date prévue.

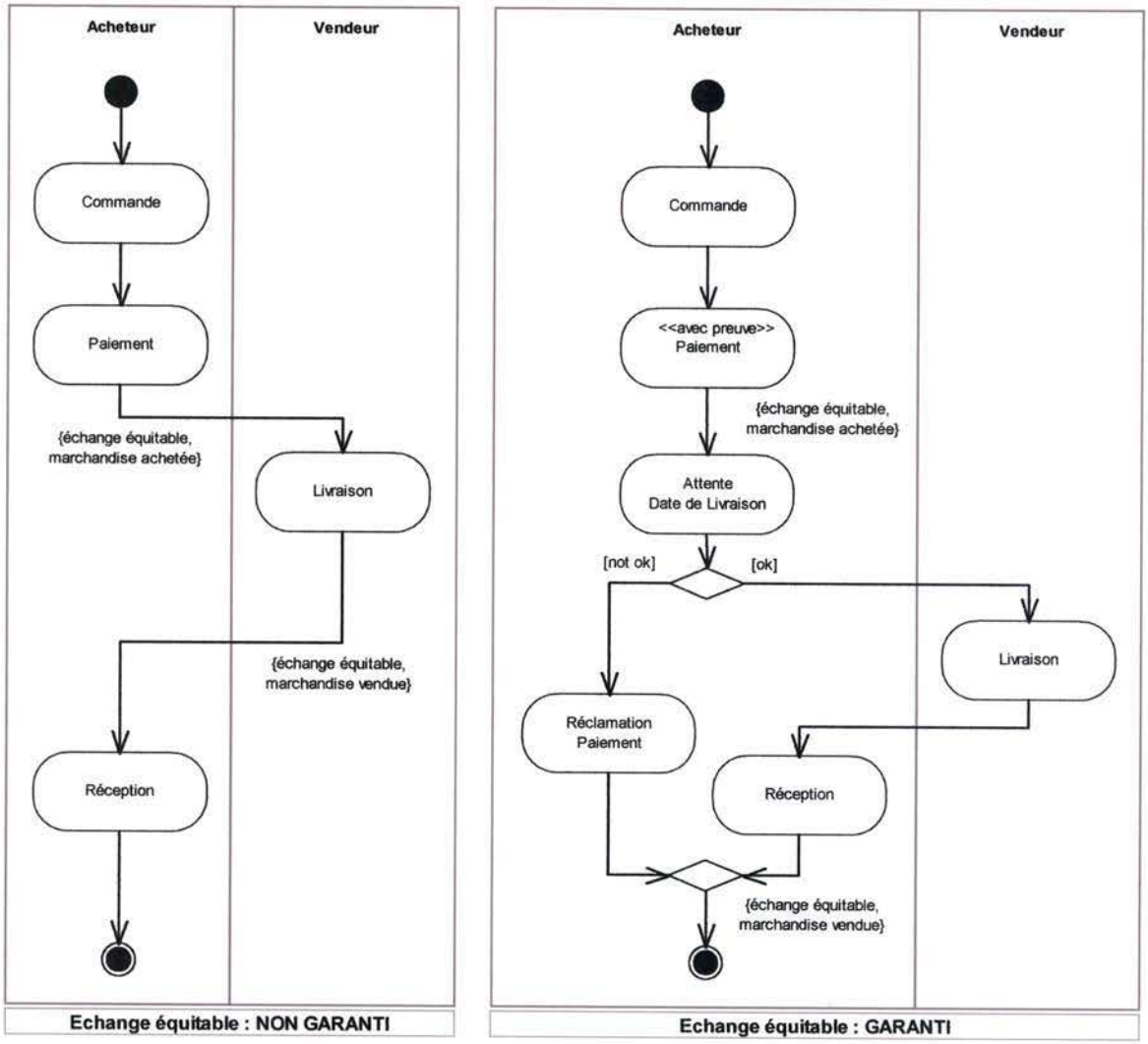


FIG. 48 – UMLSEC : DIAGRAMME D'ACTIVITE (EXEMPLE)

5.2.4. DIAGRAMME DE SEQUENCE

En matière de sécurité, les *diagrammes de séquence* permettent de spécifier avec précision les interactions critiques entre une ou plusieurs entités. Ils sont donc particulièrement utilisés pour décrire les **protocoles de sécurité**.

A l'aide d'un exemple (extrait de [JURJENS 01]), nous allons montrer comment il est possible de détecter une erreur dans un protocole grâce à ce type de diagramme.

L'idée de cet exemple est la suivante : une entité appelée *req* doit échanger une clé partagée K_M dans le but d'envoyer à une autre entité, appelée *grd*, une clé K_S protégée à l'aide de K_M. Ensuite, les objets signés avec la clé K_S pourront avoir accès à l'objet gardé par *grd*.

Pour cet exemple, nous supposons que la clé K_S doit être mise à jour régulièrement et qu'il est plus efficace d'utiliser une clé symétrique K_M pour protéger K_S (plutôt que la clé publique de *grd*).

L'identité de *req* est considérée comme connue et est liée à sa clé publique dans un certificat *cert* signé avec la clé K_C d'une autorité de certification.

Répondant à une demande *cert*(), l'objet gardé *grd* envoie un certificat auto-signé contenant sa clé publique K. En retour, l'objet *req* envoie la clé symétrique K_M signée avec sa clé privée K_r (correspondant à la clé publique de son certificat *cert*) et chiffrée avec K ainsi que son certificat *cert*.

Finalement, l'objet gardé *grd* reçoit la clé K_S chiffrée par K_M et peut autoriser l'accès aux objets signés avec la clé K_S.

Le diagramme de séquence ci-dessous traduit l'échange de message entre *req* et *grd* :

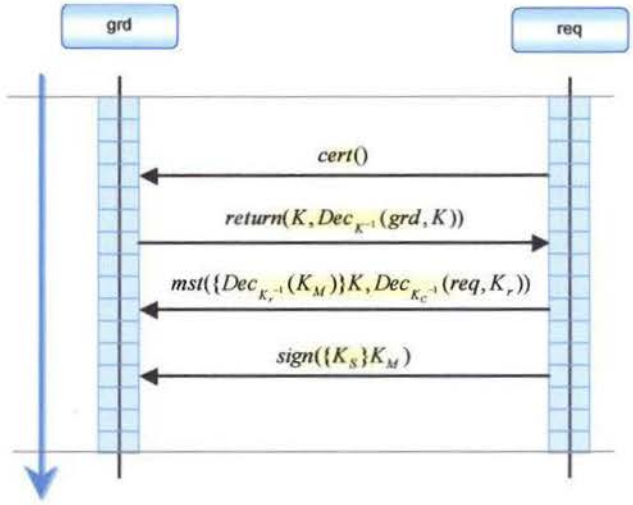


FIG. 49 – UMLSEC : DIAGRAMME DE SEQUENCE (PROTOCOLE DE SECURITE – EXEMPLE)

Malheureusement, ce mécanisme de contrôle d'accès contient une erreur : un adversaire A intercepte la communication entre *req* et *grd* et modifie les messages échangés. Il découvre la clé K_M et arrive ainsi à faire accepter par *grd* une clé K_S qu'il a choisie.

La figure ci-dessous représente la partie critique de l'échange de message correspondant à cette attaque :

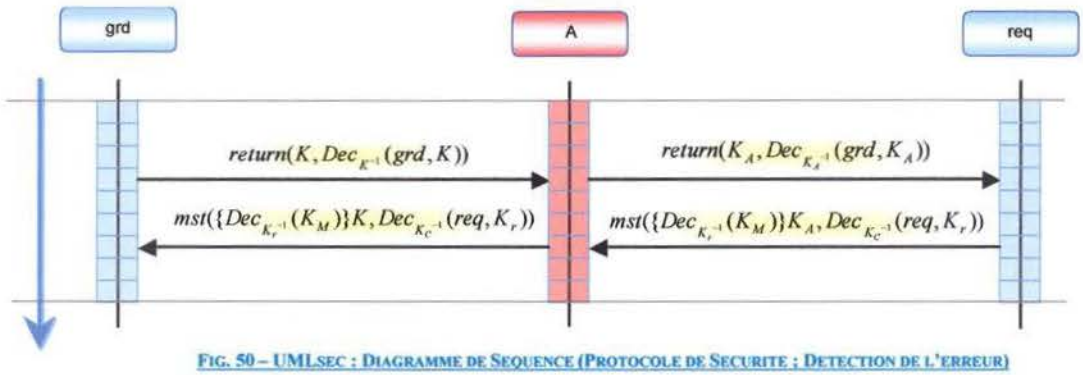


FIG. 50 – UMLSEC : DIAGRAMME DE SEQUENCE (PROTOCOLE DE SECURITE : DETECTION DE L'ERREUR)

Ce dernier diagramme de séquence permet de mettre l'erreur en évidence.

Notons tout de même que cette attaque est rendue possible par l'utilisation d'un certificat **auto-signé** que l'adversaire A a pu remplacer sans difficulté par un faux certificat.

5.2.5. DIAGRAMME DE CLASSE

Le diagramme de classe peut être utilisé pour définir la structure statique d'un système.

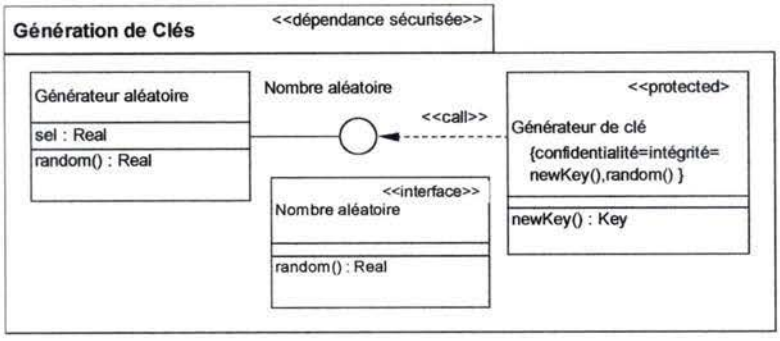


FIG. 51 – UMLSEC : DIAGRAMME DE CLASSE (EXEMPLE)

Cette figure correspond au diagramme de classe d'un générateur de clé. Nous avons déjà évoqué un générateur de ce type dans la description des sockets sécurisés SSL (voir *3.4. Zoom 3 : Secure Socket Layer*). Ce générateur de clé possède une méthode *newkey()* qui renvoie une clé permettant de garantir la confidentialité et l'intégrité de certaines données. D'un autre côté, le même diagramme montre que ce générateur de clé fait appel à un générateur de nombres aléatoires qui ne garantit ni la confidentialité, ni l'intégrité (par convention, l'absence de mention signifie que cette classe n'apporte pas ces garanties). Cet appel est spécifié par le stéréotype « *call* » entre les deux classes.

Par conséquent, ce schéma permet de montrer que globalement, l'absence de spécification sur le générateur de nombre aléatoire rend impossible la garantie de confidentialité et d'intégrité sur le générateur de clé. Ce modèle permet donc de détecter des omissions de ce type.

D'autre part, le diagramme de classe peut être utilisé dans Java pour définir la protection au niveau objet. On considère qu'à chaque classe du diagramme correspond une classe Java.

La notation UMLsec permet alors de modéliser quels seront les objets signés, scellés, ou gardés et de préciser les clés ou l'objet *Guard* concernés (voir *3.5. Zoom 4 : Objets signés, scellés, gardés*).

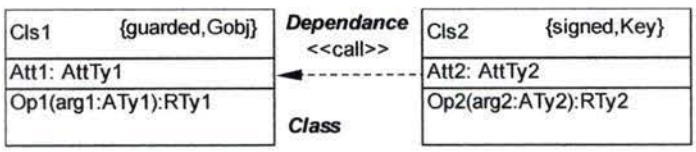


FIG. 52 – UMLSEC : OBJETS SIGNES, SCELLES, GARDES

5.2.6. DIAGRAMMES D'ÉTATS

Le diagramme d'états permet de décrire le comportement dynamique d'un objet en fonction des événements qu'il reçoit. Soulignons que son utilisation est tout particulièrement indiquée pour les objets gardés dans l'environnement Java.

La figure ci-dessous donne un exemple de ce type. Supposons que l'accès à une ressource n'est possible que par une applet signée par le site *Finance*, avec un maximum de 5 accès par semaine. La variable *thisWeek* permet de compter le nombre d'accès par semaine et *weekLimit* est vrai si la limite de 5 accès n'est pas atteinte.

Partant de l'état initial *WaitReq*, l'objet gardé reçoit une demande *checkGuard*. Si le résultat du test vérifie les conditions *[origin=signed=Finance, weekLimit]*, la variable *thisWeek* est incrémentée et la référence à la ressource est renvoyée, sinon une exception de sécurité est levée.

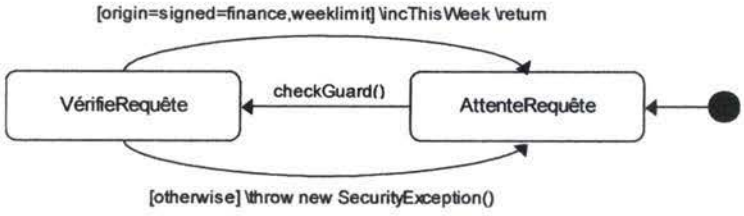


FIG. 53 – UMLSEC : DIAGRAMME D'ETATS-TRANSITIONS (EXEMPLE)

5.2.7. DIAGRAMME DE DEPLOIEMENT

Les diagrammes de déploiement décrivent la couche physique du système. Ils sont pertinents d'un point de vue sécurité car ils donnent la localisation des différents composants du système. D'une part ils permettent d'indiquer le type de lien physique entre les nœuds du système et, d'autre part, le type de communication entre les différents composants.

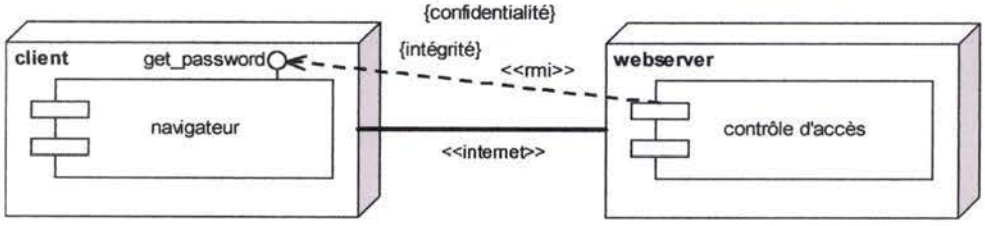


FIG. 54 – UMLSEC : DIAGRAMME DE DEPLOIEMENT (EXEMPLE)

Dans l'exemple ci-dessus, le lien physique est précisé par le stéréotype « *internet* » et le type de communication entre le composant *webserver* et le *client* par le stéréotype « *rmi* ». De plus, l'échange de données entre ces composants doit garantir les contraintes *{confidentialité}* et *{intégrité}*.

5.2.8. DISCUSSION DU MODELE UMLSEC

En résumé, UMLsec étend la plupart des diagrammes UML standards pour mettre en évidence les différents aspects de la sécurité d'un projet :

- **Diagramme de Use Cases** : permet d'exprimer les exigences de sécurité ;
- **Diagramme d'activité** : permet la description du contrôle d'un échange sécurisé ;
- **Diagramme de classe** : montre à quel niveau se situent les échanges de données, ce qui permet de préserver leur sécurité par des moyens adéquats ;
- **Diagramme de séquence** : est utilisé pour décrire les sections critiques d'une interaction sécurisée (protocole de sécurité ou section critique d'un système distribué) ;
- **Diagramme d'états** : permet de voir si la sécurité est préservée à l'intérieur d'un objet
- **Diagramme de déploiement/composants** : permet de vérifier si les exigences de sécurité sont remplies au niveau physique.

Les avantages de l'approche proposée par Jan Jürjens sont multiples :

- il étend la notation UML qui est devenue un standard *de facto* pour la modélisation des systèmes orientés-objet ;
- il utilise les extensions standards d'UML, ce qui permet de tracer les différentes vues de la sécurité d'un système avec les outils CASE habituels ;
- il utilise les différents diagrammes UML pour évaluer les points critiques de sécurité sous différents angles de vue ;
- il propose une approche modulaire favorisant la réutilisation du code relatif à la sécurité.

[JURJENS 01] met en évidence les diagrammes les plus pertinents pour le développement d'applications Java sécurisées avec UMLsec : le **class diagram** permet de visualiser les dépendances entre les différents objets et de choisir quels seront les objets signés, scellés ou gardés.

Le fonctionnement interne d'un objet gardé peut être illustré par un **statechart diagram**.

Le **diagramme de déploiement/composants**, quant à lui, fournit une vue d'ensemble en montrant la localisation physique des nœuds du système et la place des composants (la localisation des composants correspond notamment aux sources de code utilisées dans les permissions). C'est sur ce diagramme que seront précisés les types de liens (par exemple internet) et les types de communication (par exemple RMI).

Un autre volet des travaux de Jürjens concerne la création d'une sémantique **formelle** basée sur la notation UML Actuellement, étant donné qu'il n'existe aucun accord sur une telle sémantique pour UML, il utilise une sémantique formelle personnelle, adaptée à ses recherches, pour évaluer les diagrammes UML et y détecter les faiblesses éventuelles. Il inclut notamment les attaques possibles sous la forme d'un adversaire dans les statecharts diagrams et les diagrammes de déploiement.

Cette approche est séduisante car détecter les faiblesses d'un système de sécurité à l'aide d'une sémantique formelle permettrait de valider la cohérence d'un scénario (correspondant par exemple à un protocole de sécurité). En outre, cette validation pourrait se faire au moyen d'un outil automatique.

Notons toutefois qu'un tel raisonnement formel repose essentiellement sur le **modèle utilisé**. Un modèle reste une simplification de la réalité et correspond à ce que nous connaissons de la réalité à un moment donné. La validation obtenue est donc limitée aux composantes de ce modèle. Or, en matière de sécurité, les attaques surviennent généralement là où on ne les attendait pas…

Néanmoins l'approche formelle ne doit pas être rejetée car si elle ne permet pas de détecter toutes les faiblesses d'un système, elle peut, malgré tout, détecter les incohérences d'un modèle, ce qui constitue déjà un outil appréciable.

6. GUIDE DE SÉCURISATION D'UN PROJET JAVA

[BROSE 01] • [MAIWALD 01] • [McGRAW 98] • [NIEMEYER 00] • [VIEGA 01]

6.1. INTRODUCTION

Il ne s'agit pas ici de décrire une méthode de sécurisation d'un projet mais plutôt de donner un guide pour améliorer la conception de logiciels sécurisés. Bien que ce mémoire s'intéresse particulièrement aux techniques de sécurité Java, il faut souligner que ce guide est conçu pour s'appliquer à tout projet de développement d'applications orientées objets. Toutefois, les propositions techniques du Guide liées aux phases d'implémentation sont basées sur les mécanismes originaux de Java en matière de sécurité que nous avons décrits dans la première partie.

Il est à noter également que d'autres techniques qui sortent du cadre de ce mémoire doivent être mises en œuvre lorsque l'on considère la sécurité d'un système dans sa globalité. Ceci inclut notamment certaines techniques de construction du réseau, que nous citerons sans les détailler : segmentation du réseau, firewalls, zones démilitarisées, etc.

Pour commencer, il est important de bien prendre conscience qu'une démarche de sécurité ne vise nullement à résoudre définitivement les problèmes liés à la sécurité, mais bien à chercher à réduire le nombre de possibilités ou la facilité avec laquelle le comportement ou les ressources d'un système pourraient être détournés ou entravés.

En outre, un principe général doit être appliqué dans l'analyse des fonctionnalités de sécurité d'une application : chaque partie de sécurité identifiable doit être sécurisée indépendamment des autres parties, c'est-à-dire sans se baser sur la sécurité établie pour les parties avec lesquelles elle est liée.

Cette relative autonomie de sécurité permet un développement indépendant et une consolidation naturelle des éléments sécurisés entre eux. Ce principe garanti en outre une meilleure cohérence finale, particulièrement dans le cadre du développement d'une application de taille importante pour laquelle la modularité et l'indépendance des modules revêt un caractère vital.

Il est aussi important de se rendre compte que le niveau de sécurité atteint dans un système dépendra de son maillon le plus faible. Sécuriser une partie d'une architecture est insuffisant, voire inutile. Par analogie, ce n'est pas utile de blinder une porte si elle ne ferme même pas à clé !

C'est pourquoi la sécurité doit être considérée comme une chaîne : les dépendances d'un système de sécurité doivent elles aussi être sécurisées. Ainsi, dans l'exemple de *class diagram* utilisé dans la partie UMLsec (voir *section 5.2.4*), nous avons vu un générateur de clé faisant appel à un générateur de nombre aléatoire non sécurisé, ce qui rendait caduques les garanties de confidentialité et d'intégrité du générateur de clé.

Dans notre Guide de Conception Sécurisée, nous fédérons les principes décrits jusqu'à présent. Les étapes du guide correspondent aux grandes étapes de conception décrites dans la partie *Modélisation UML* et *Modélisation UMLsec*. Nous tenterons d'intégrer l'aspect sécurité à chacune de ces étapes, en faisant référence aux concepts et aux techniques de la première partie.

6.2. GUIDE DE CONCEPTION SÉCURISÉE

La figure ci-dessous introduit les préoccupations de sécurité à travers chaque étape de la méthode de développement en cascade :

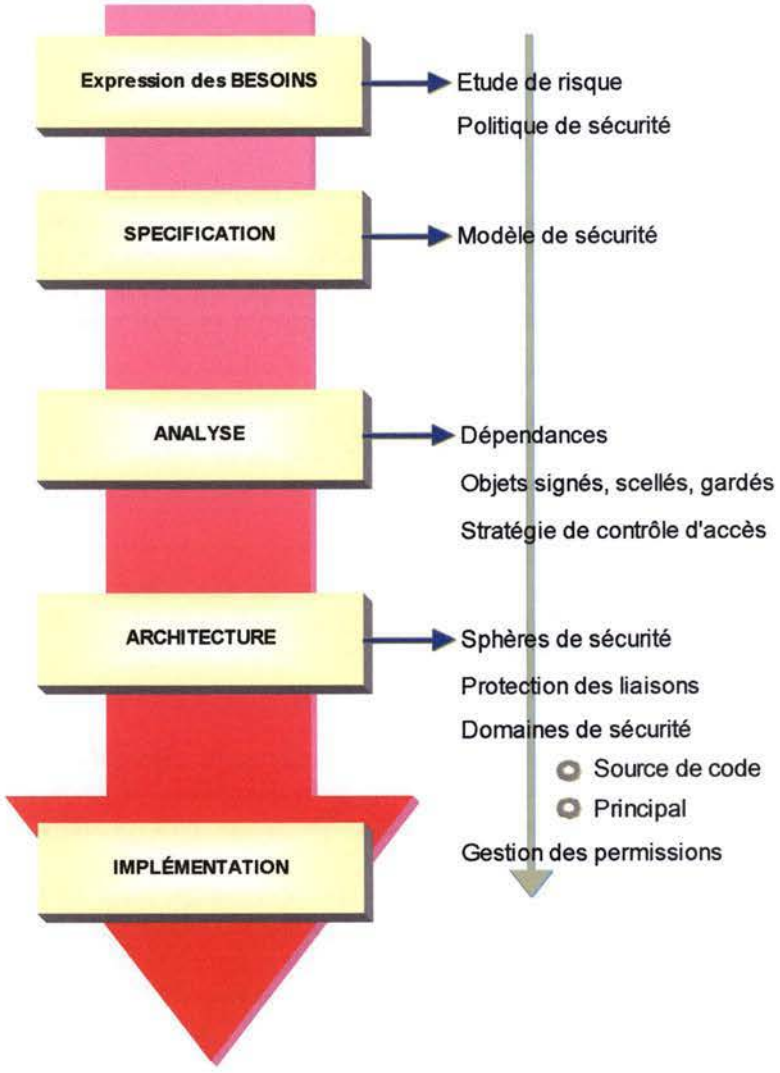


FIG. 55 – INTÉGRATION DE LA SÉCURITÉ DANS LA MÉTHODE DE DÉVELOPPEMENT

La partie *Modélisation UML* nous a montré que les applications informatiques doivent être construites de façon méthodique par des transformations successives au départ de spécifications fonctionnelles. Ainsi, le processus que nous avons décrit fait progressivement basculer les considérations du fonctionnel à l'objet, la phase charnière étant le passage de la phase de Spécification à la phase d'Analyse au travers des diagrammes de séquence.

Les applications ainsi construites constituent un développement ininterrompu des spécifications fonctionnelles, et non pas une construction indépendante. **Toute la difficulté du processus d'analyse réside en réalité dans la cohérence à maintenir en passant d'une transformation à l'autre**.

Les mêmes considérations peuvent être tenues s'agissant des techniques de sécurité dans ces applications. En effet pour être efficaces, les mécanismes de sécurité ne peuvent être insérés aveuglément dans une application : ils doivent donc être pris en compte dès les premières phases du processus de conception général.

Notre objectif n'est pas d'entrer dans le détail de chacune de ces étapes pour en donner une vue exhaustive. Au contraire, notre guide de conception vise à donner une démarche générale permettant de faciliter la mise en œuvre de la sécurité sans nécessiter une grande phase d'apprentissage. Nous rejoignons en cela les objectifs de pédagogie et de ligne de conduite énoncés dans l'introduction.

6.2.1. EXPRESSION DES BESOINS

Comme nous l'avons vu dans le chapitre consacré à la méthodologie basée sur la notation UML (voir section *5.1. Modélisation UML* et suivantes), la première étape d'un processus de conception d'application est la définition des exigences (expression des besoins). Parmi ces exigences, nous mettrons en évidence celles liées à la sécurité, c'est-à-dire :

- l'identification des ressources sensibles,
- la protection des ressources sensibles,
- l'identification pour les accès ou les opérations,
- l'audit du système,
- la disponibilité des informations.

En réponse à ces exigences, nous retrouvons les services de sécurité exposés dans le chapitre *2. Sécurité de l'Information* : confidentialité et intégrité (authenticité de contenu), vérification de l'identité, non-répudiation, permissions d'accès, audit, disponibilité du service.

Le niveau de sécurité visé doit également faire l'objet d'une **étude du risque**. **Le risque est la probabilité qu'une menace exploite une vulnérabilité du système**. La notion de risque implique le besoin de *protection* d'une ressource. Cette protection est liée à la notion de *menace* sur cette ressource, autrement dit à la *perte potentielle* consécutive à un défaut de protection. L'évaluation de l'importance de cette perte potentielle détermine le degré de **sensibilité** de la ressource.

Une menace est donc une action ou un événement susceptible de violer la protection d'un système d'information. La cible d'une menace est une ressource de ce système. Il est important de distinguer les attaquants potentiels, afin de mieux déterminer les moyens dont disposent ces attaquants pour atteindre leur cible (types d'attaques possibles).

Les contre-mesures prises pour s'opposer à la menace placent le système à un certain degré de protection. L'inadéquation de ces mesures détermine la *vulnérabilité* du système.

Le rapport entre la vulnérabilité et la menace est illustré dans le schéma ci-dessous (extrait de [MAIWALD 01]). Ce schéma montre clairement qu'en l'absence de vulnérabilité ou de menace, il n'y a pas de risque.

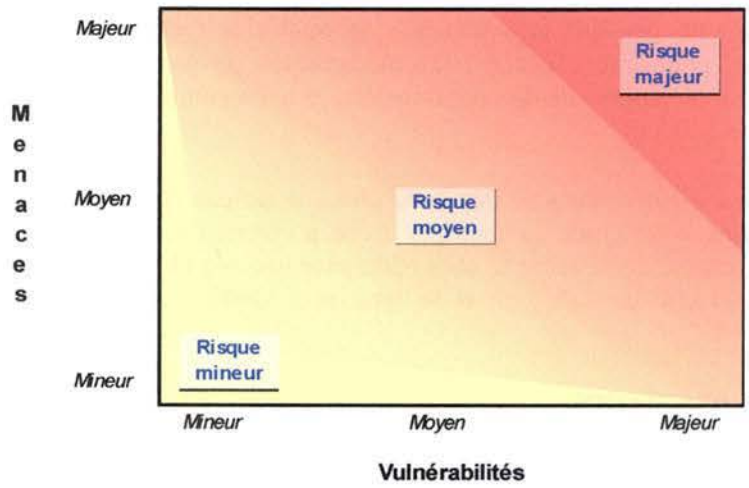


FIG. 56 – RISQUE = MENACE * VULNÉRABILITÉ

Il faut souligner que les mesures prises pour assurer la sécurité d'un système ne sont pas seulement dictées par des impératifs techniques. En effet, ces contre-mesures induisent des coûts, en temps et en ressources, qui dépendent des objectifs fixés en matière de sécurité et des techniques mises en œuvre pour y parvenir. Un facteur important à prendre en compte lors de la conception sécurisée d'un système est donc de nature économique : quel est le coût d'une contre-mesure par rapport au coût potentiel de la vulnérabilité qu'elle doit annuler ? Et quel est le coût que doit supporter un éventuel attaquant par rapport au gain que lui procurerait la réussite de son action ? *Quel est le coût du risque par rapport au coût de la diminution du risque ?*

En général, plus on veut diminuer le risque, plus l'effort est important et plus les moyens requis sont conséquents. **Gérer la sécurité, c'est donc gérer le risque, c'est-à-dire savoir placer le seuil en dessous duquel le risque devient acceptable compte tenu des moyens disponibles**.

Cette notion de risque est intimement liée aux menaces identifiées et à la vulnérabilité du système, compte tenu des contre-mesures effectives et des failles présentes, quelle que soit leur origine technique ou économique. Il est crucial de bien évaluer les risques encourus par un système. Ce risque peut être jugé de plusieurs niveaux : mineur, moyen, majeur. Le niveau de risque correspond généralement à une traduction **économique** des conséquences techniques d'une faille.

Dans le cadre d'applications distribuées, le vecteur évident des attaques à l'encontre d'un système est le réseau informatique autour duquel il est construit. Dans la première partie, nous avons évoqué les problèmes de sécurité qui pouvaient y survenir, c'est-à-dire les actions qu'un pirate peut tenter pour atteindre une cible, et quelles sont les principales motivations qui guident ces actions (voir *2.2.2. Sources d'incidents de sécurité sur un réseau*).

En résumé, la première phase de conception de l'application permet d'exprimer les propriétés externes du système, **indépendamment** de toute description technique ou fonctionnelle. Il s'agit de prendre en compte les risques auxquels le système est exposé ainsi que les besoins en matière de sécurité pour définir une **politique de sécurité**, c'est-à-dire, un compromis entre un **niveau de risque acceptable** et le coût des contre-mesures à mettre en place pour y parvenir.

6.2.2. SPÉCIFICATIONS FONCTIONNELLES

La vue externe de l'application obtenue dans la première phase va être complétée par une dimension fonctionnelle. Il s'agit donc de décrire les fonctionnalités réelles du système, au travers des *use cases* et des *diagrammes de use cases* (voir *5.1.2. Spécification*).

A ce propos, nous avons trouvé l'approche de [BROSE 01] particulièrement intéressante : il utilise les use cases pour la spécification des exigences liées à la sécurité. A partir du moment où le use case décrit quel acteur a besoin d'accéder à quel élément de l'application pour remplir sa fonction, il définit implicitement une police d'accès pour ces éléments.

De plus, cette police d'accès adhère au **principe de moindre privilège**. Cela implique que tout élément du système doit être considéré comme protégé (c'est-à-dire limité à des accès explicitement représentés), et que tout accès dans le système qui n'est pas explicitement décrit doit être considéré implicitement comme interdit.

[BROSE 01] propose d'attacher à chaque association acteur-action une note textuelle décrivant ce qui est autorisé ou refusé à l'acteur.

L'exemple ci-dessous illustre ces principes. Il s'agit d'un agenda électronique permettant de gérer des rendez-vous et des réunions. Tous les utilisateurs du système peuvent visualiser les entrées du calendrier. Le propriétaire du calendrier peut autoriser sa secrétaire à créer et modifier des entrées dans l'agenda mais il est le seul à pouvoir supprimer une entrée de l'agenda. La secrétaire est la seule à pouvoir réserver le local où se déroulera la réunion.

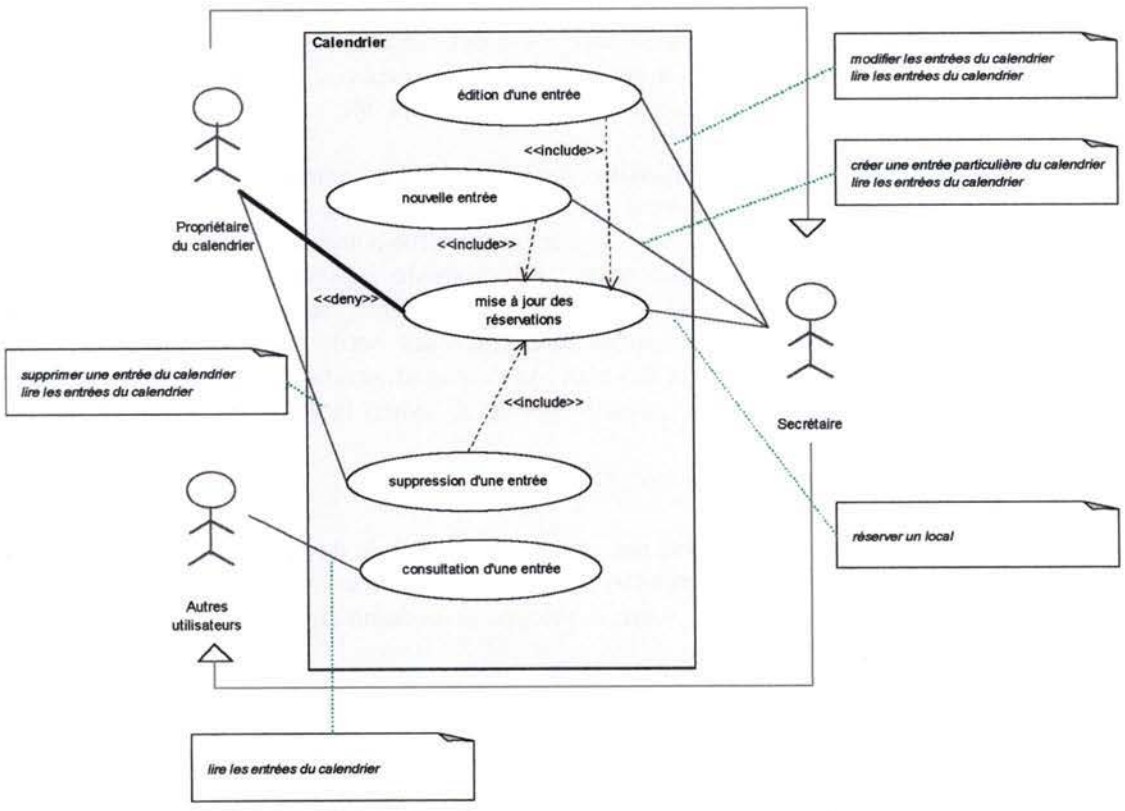


FIG. 57 – DIAGRAMME DE USE CASES AVEC IDENTIFICATION DES ACCÈS.

Comme illustré dans cet exemple, la notation de [BROZE] offre également la possibilité d'ajouter sur le diagramme de use cases une spécification « *deny* » qui correspond à un **use case interdit**, c'est-à-dire un accès explicitement non autorisé. En effet, il est plus courant d'exprimer les règles de sécurité de façon **négative**, sous forme d'interdiction : par exemple, certaines personnes ne sont pas autorisées à lire tel fichier, une opération ne peut pas être effectuée par certains clients, certaines informations ne peuvent jamais quitter une organisation, etc.

L'utilisation de règles de sécurité positives et négatives est plus proche de la gestion courante des permissions par un administrateur. Le diagramme de use cases ainsi obtenu pourrait alors servir de base pour établir les règles de sécurité de l'application.

Nous voudrions souligner deux avantages essentiels de cette approche, liés au fait qu'elle se situe lors de la phase de spécifications fonctionnelles.

Le premier avantage est que le niveau conceptuel auquel on se situe permet de se détacher de toute considération technique, ce qui est évidemment très difficile lorsqu'on se trouve à un niveau objet. **La sécurité de base doit permettre à l'administrateur de gérer les accès aux ressources sans connaissance particulière sur l'implémentation de l'application.**

L'autre avantage est qu'il est possible à travers le diagramme de use cases de décrire les aspects de permissions dans un langage proche de celui des utilisateurs du système, donc de valider ce modèle de permissions avec les personnes intéressées.

La description fonctionnelle de l'application doit conduire à la construction du **modèle de sécurité** de l'application. Celui-ci doit indiquer les ressources que l'on désire protéger (informations stockées et flux d'informations), contre quelles menaces, et avec quel degré de sensibilité. Ce modèle doit être évalué à la lueur de la politique de sécurité qui a été définie lors de l'analyse des besoins, pour déterminer les mesures à prendre pour éliminer ou réduire les menaces identifiées dans le modèle de sécurité et parvenir à protéger ces ressources avec un risque acceptable.

La politique de sécurité comporte un **axe passif** et un **axe actif**. La sécurité passive consiste à fortifier le système contre des attaques éventuelles ; c'est celle qui est évoquée généralement lorsqu'on parle sans autre précision de « sécurité du système ». La sécurité active consiste à mettre en place des outils de détection et d'analyse des attaques effectives. Elle consiste à surveiller les moyens passifs de protection, à détecter les attaques ou les défaillances du système, et à permettre des actions de correction ou de réponse efficaces. Ceci inclut notamment des outils d'analyse adéquats – permet d'assurer le *service de sécurité d'audit*, et des moyens de reconfiguration dynamiques permettant au système de muter en fonction du contexte (panne) – permet d'assurer la *disponibilité du service*.

6.2.3. ANALYSE ORIENTÉE OBJETS

Nous abordons à présent la phase d'analyse qui va nous permettre de passer d'une vue fonctionnelle à une vue orientée-objet. Ce passage important repose sur la transformation des use cases en diagrammes de séquence ainsi que nous l'avons développé précédemment (voir *5.1.3 Analyse*).

Ce diagramme d'interactions fait intervenir :

- les acteurs : déjà définis dans les use cases ;
- les interfaces : elles permettent la communication entre un acteur et le système, comme par exemple une boîte de dialogue, un formulaire, un menu, un message d'alerte, etc. ;
- les contrôleurs : ce sont des composants dotés d'une certaine « intelligence » ;
- les *repositories* : ils permettent de stocker de l'information et correspondent généralement à un élément de base de données.

Le diagramme de séquence décrit les interactions entre les différentes composantes du use case, ainsi que les messages qui sont échangés.

Le mécanisme qui va permettre de passer d'une vue fonctionnelle à une vue orientée objets est également applicable aux permissions. Nous avons tout d'abord décrit les permissions à l'aide d'une note textuelle sur le diagramme de use cases, à un niveau purement fonctionnel. Ensuite, les diagrammes de séquence nous permettent d'identifier les accès nécessaires entre les objets eux-mêmes (quel objet a la permission d'appeler une méthode d'un autre objet).

L'utilisateur, quant à lui accède aux données du système au travers des interfaces ; l'interface permet d'envoyer des requêtes à un contrôleur ; le contrôleur communique éventuellement avec d'autres contrôleurs, ou accède à une des données via un *repository*.

C'est ainsi que seront mis en évidence les différents accès aux bases de données, et le type d'accès requis (création, lecture, écriture, suppression, etc.). Ce mécanisme fournit implicitement une **stratégie de contrôle d'accès** à ces données, qui adhère toujours au principe de moindre privilège, et qui constitue un raffinement de la police d'accès déduite de la phase précédente (voir *6.2.2. Spécifications fonctionnelles*).

L'agrégation des objets, des messages et des paramètres décrits dans les diagrammes de séquence permet d'obtenir presque mécaniquement une description des composants actifs du système (par l'agrégation de tous les messages des diagrammes de séquence pour ces composants). Ces derniers sont la base du *diagramme de classe*, qui met en évidence les liens conceptuels (les *dépendances*) entre les objets. Le diagramme de classe fournit une **vue microscopique** du système.

L'analyse des dépendances fait ressortir les échanges d'informations entre objets. Les dépendances existent pour différentes raisons : une classe envoie un message à une autre, une classe fait partie des données d'une autre classe, une classe en mentionne une autre comme paramètre d'une opération.

A la lumière de la stratégie de contrôle d'accès aux données, apparaissent les messages qui concernent des données « sensibles » (qui nécessitent une protection conforme à la politique de sécurité). Ces messages traduisent des accès aux objets par appel à leurs méthodes. La sécurité des accès au niveau de ces objets peut être traduite par le concept Java des objets signés, scellés, gardés (voir *3.5. Zoom 4 : Objets signés, scellés, gardés*).

Au niveau de l'objet lui-même, les objets signés fournissent le service d'intégrité, les objets scellés celui de la confidentialité, et les objets gardés le service des permissions d'accès. La dimension distribuée des applications renforce l'utilité de ces mécanismes ; c'est pourquoi ils pourront être affinés à l'aide du diagramme de déploiement, lors de la phase de conception de l'architecture décrite dans la section suivante.

Notons encore un contexte particulier où les diagrammes de séquence se montrent spécialement adaptés, dans le cas où le système analysé est un mécanisme de sécurité à part entière. Ils permettent en effet de mettre clairement en évidence les échanges entre les partenaires intervenant dans toutes les phases d'un protocole à négociation : identification et authentification, algorithme de distribution des clés, etc. Cette utilisation a déjà été illustrée lors de la description du *handshake protocol* de SSL (voir *2.3.7. Secure Socket Layer*), et surtout lors de la mise en évidence de la validation formelle d'un protocole dans la méthodologie UMLsec (voir *5.2.4. Sequence Diagram*).

6.2.4. CONCEPTION DE L'ARCHITECTURE

L'architecture logique regroupe les classes en packages selon les dépendances entre objets. L'architecture physique est basée sur la localisation physique des composants. Le diagramme de déploiement montre les liens physiques entre les nœuds (localisations physiques), et les dépendances entre les composants logiques. Ces liens et ces dépendances nécessitent une attention particulière du point de vue de la sécurité.

Nous appellerons **sphère de sécurité** [1] un ensemble de ressources informatiques avec des besoins de sécurité homogènes. Les ressources d'une sphère communiquent entre elles et peuvent également communiquer avec les ressources d'une autre sphère. Ces communications s'effectuent via des liens physiques ou logiques.

Les connexions physiques traduisent des connexions réseaux, qu'il s'agisse d'un réseau local ou d'un réseau public tel Internet. Les connexions apparaissent clairement dans le diagramme de déploiement, comme étant les liaisons de communication entre les localisations physiques (les nœuds du schéma ; voir *5.2.7. Diagramme de déploiement*). Le diagramme de déploiement fournit une **vue macroscopique** du système.

D'autre part, des dépendances existent entre les composants à l'intérieur d'un ou de plusieurs nœuds. Nous distinguerons par conséquent la technologie de support « physique » utilisée pour le transport des données de la technologie de communication utilisée sur ce support (liaison logique).

Nous dissocierons plusieurs niveaux de sécurité à appliquer aux liens physiques ou logiques, selon le degré de confiance qu'on peut leur accorder :

1. **Public externe** : ce niveau correspond à l'ensemble des ressources accessibles depuis l'extérieur du système, typiquement via un réseau public de type Internet.
2. **Public interne** : c'est la vision interne des ressources informatiques tournées vers l'extérieur : base de diffusion, serveurs Webs, serveurs de mails, etc.
3. **Privé général** : ce niveau correspond au réseau intérieur du système ;
4. **Privé local** : nous distinguerons ici un sous-réseau constituant une sphère ; par exemple le sous-réseau des serveurs de base de données, le sous-réseau des serveurs applicatifs, etc. La distinction s'effectue entre la communication des éléments du sous-réseau entre eux et la communication avec d'autres sous-réseau.
5. **Privé local sensible** : ce niveau met en évidence un sous-réseau ou un élément particulier d'un sous-réseau identifié comme étant particulièrement sensible du point de vue de la sécurité.

Les sphères considérées, le type de liaison et les niveaux de sécurité associés aux sphères détermineront les choix à opérer en matière de protection. Lorsque des informations transitent entre des sphères de niveaux différents, des moyens de protection doivent être envisagés pour assurer le niveau de sécurité requis, par exemple en matière de confidentialité.

Les mécanismes de sécurité à mettre en place dépendent de la confiance que l'on accorde aux couches « inférieures ». Notons qu'il faut également tenir compte de la confiance accordée à l'environnement d'exécution du système : système d'exploitation, réseau, middleware, etc.

La protection des liaisons physiques et logiques dépendra du modèle de sécurité établi lors de la phase de spécification fonctionnelle, et du modèle de sécurité qui en a résulté. En effet, c'est en fonction de la sensibilité fonctionnelle établie à ce niveau que l'on pourra améliorer l'interprétation du niveau de

[1] La littérature parle parfois de *domaine de sécurité*. Toutefois, nous préférons parler de *sphère de sécurité* afin d'éviter de confondre avec les domaines de protection, terme consacré dans un contexte de permissions.

sécurité requis. Nous retrouvons donc ici les exigences de sécurité passives ou actives déterminées selon l'étude du risque de la première phase.

Il est important d'insister sur le fait que les choix d'implémentation de la sécurité sont directement liés aux spécifications des besoins décrites précédemment. En effet, c'est aussi en fonction de la sensibilité reconnue des informations à protéger, en fonction des vulnérabilités et des menaces identifiées et du coefficient de risque attribué qu'il faut envisager le choix des contre-mesures (voir *6.2.1. Expressions des besoins*).

Les **liaisons physiques** sont principalement concernées par les services de sécurité liés à la confidentialité et à l'intégrité (authenticité de contenu). Nous retrouverons ici le choix de la réponse cryptographique correspondant aux besoins exprimés. Il s'agira d'inventorier les principes cryptographiques adéquats (clé symétriques/asymétriques, taille des clés, condensés de messages, etc.) et les algorithmes disponibles (DES, RSA, SSL, etc.) – voir *2.3. Protocoles de sécurité.*

Les **liaisons logiques**, quant à elles, sont essentiellement concernées par les services de sécurité liés à l'authentification (vérification de l'identité, authentification de l'origine des messages ou du code), aux permissions d'accès, et à la non-répudiation. De la même façon, la sélection des principes de sécurité adéquats (certificats, signature numérique) et des algorithmes disponibles (norme X509) doivent fournir une solution utile.

Les liaisons logiques sont également concernées par les services de confidentialité et d'intégrité, indépendamment de la protection mise en place au niveau physique. En effet, il n'est peut être pas suffisant d'assurer ces protections au niveau communication ; il peut être pertinent de les mettre en œuvre au niveau des objets eux-même. L'utilité de sécuriser au niveau objets a été mise en évidence lors de la phase d'analyse orientée objets (voir section précédente).

Par exemple, dans une application Java, la liaison physique serait assurée par TCP/IP, et la liaison logique utiliserait la technologie RMI, ou une autre architecture d'objets distribués comme CORBA (voir *4. Architectures Distribuées*). La liaison physique pourrait être sécurisée via le protocole SSL (voir *2.3.7. Zoom 3 : Secure Socket Layer*), alors que la liaison logique pourrait utiliser des objets scellés (voir *3.5. Zoom 4 : Objets signés, scellés, gardés*). Si une protection au niveau des objets ne se justifie pas, il peut être suffisant de sécuriser la liaison physique en appliquant RMI sur SSL.

Il ne faut cependant pas perdre de vue que, dans le choix d'une solution aux besoins des liaisons physiques et logiques, interviendra le type de plate-forme de développement prévu. Dans le cadre de ce mémoire, il s'agit bien entendu de Java. Il faut donc définir une réponse technique adaptée à l'architecture de sécurité de Java. La connaissance des caractéristiques de sécurité de Java et surtout des moyens originaux du langage est une condition nécessaire à un choix judicieux.

C'est ainsi par exemple que l'on appréciera l'intégration du protocole de sécurité SSL pour la sécurisation des liaisons physiques (voir *3.3. Zoom 3 :Secure Socket Layer*). Ou encore la possibilité de définir un chargeur de classe personnalisé qui permet notamment de répondre à un besoin d'authenticité de code (voir *3.1. Zoom 1 : Chargeur de classe*). Ou, enfin, l'ensemble des classes composant le Contrôleur d'accès, permettant une granularité très fine dans la gestion des permissions (voir *3.2. Zoom 2 : Gestionnaire de sécurité et Contrôleur d'accès*).

La phase d'analyse nous a fournit une stratégie de contrôle d'accès basée sur les objets. En pratique, la protection d'accès de chaque objet pris individuellement est ingérable compte tenu de la propension de l'orienté objets à engendrer une multitude d'objets différents (voir *2.4.2. Sécurité des objets et problèmes particuliers*).

Il est donc indispensable de gérer l'accès aux ressources en agrégeant les permissions de nature similaire. Les permissions agrégées constituent un **domaine de sécurité**, soit un ensemble de ressources avec une stratégie de sécurité homogène. Les domaines de sécurité peuvent être définis hiérarchiquement en sous-domaine ou en domaines fédérés (voir *2.1.4. Permission d'accès*).

Les domaines de sécurité seront basés sur la stratégie de contrôle d'accès définie lors de l'analyse orientée objets. Ils respecteront donc le principe de moindre privilège, en définissant un ensemble minimal de permission qui pourra ensuite être étendu.

L'agrégation des permissions peut s'effectuer autour d'une source de code ou autour d'un principal, autrement dit selon l'origine de la demande d'accès ou selon l'identité ou le rôle du demandeur.

Notons que, d'un point de vue implémentation, la notion de rôle n'existe pas en Java, mais qu'elle peut être vue comme un domaine de permission (au sens Java). En outre, Java offre une granularité fine en permettant jusqu'à la création d'une permission pour une seule méthode d'un objet (cas limite) au travers des domaines de permissions.

Java permet également de créer des permissions personnalisées, et de créer des relations d'implications entre les permissions (une permission générale impliquant une permission plus spécialisée). Les possibilités très étendues de Java en matière de permissions sont décrites dans *3.3. Zoom 2 : Gestionnaire de sécurité et contrôleur d'accès*.

Notons que la **gestion des permissions** s'effectue indépendamment des structures microscopiques (classes) ou macroscopiques (déploiement) de l'application. Elle est construite sur une **vue globale** de la sécurité du système.

6.2.5. IMPLÉMENTATION ET RÈGLES DE PROGRAMMATION JAVA

L'implémentation des techniques de sécurité Java énoncées tout au long du Guide a déjà été détaillée dans la première partie. En complément, nous énoncerons ici quelques considérations directement liées à la nature du code Java, indépendantes de l'application elle-même. Même si les règles proposées s'éloignent de la démarche générale de conception de ce Guide, il nous semble intéressant d'attirer l'attention sur certains points très particuliers d'implémentation.

Nous commencerons par une première remarque qui n'est pas spécifique au langage Java, mais qui s'y applique pertinemment : il s'agit du stockage des clés de protection (ou des mots de passe) directement dans le code source, ou encore de la sécurité illusoire qui consiste à insérer le secret de sécurité dans l'algorithme lui-même. Il ne faut jamais perdre de vue qu'il est impossible d'empêcher un utilisateur disposant des outils nécessaires d'analyser le code chargé en mémoire (dans le cas de Java, dans la JVM), voire même de fabriquer sa propre JVM (dont les spécifications sont publiques) pour ce faire.

Une autre règle essentielle de sécurité à mettre en œuvre correspond à un principe fondamental de la programmation orientée objets : l'encapsulation, et la visibilité des objets et méthodes. Une variable d'instance ne peut être accédée que par des méthodes explicites de la classe. De plus, la visibilité des variables et des méthodes doit être réduite au maximum. Le principe à appliquer peut être rapproché de celui envisagé pour les permissions : partir d'une visibilité minimale et accroître uniquement celle des variables / méthodes pour lesquelles cela est nécessaire.

Attention, la visibilité par défaut d'une classe, qui est celle du package de la classe, n'est pas jugée suffisante d'un point de vue sécurité, car il est toujours possible d'étendre un package avec une classe qui aurait alors automatiquement accès aux méthodes et variables avec cette visibilité.

Il en va de même pour la déclaration de méthodes finales, qui permet d'éviter qu'une classe soit étendue : toute classe doit être déclarée finale, si elle ne doit pas explicitement être étendue. Ceci semble en contradiction avec le principe objet d'extensibilité du code et de réutilisation des objets. Toutefois, l'extensibilité est l'ennemie de la sécurité, en ce sens qu'elle fournit à un éventuel pirate des points d'entrée dans le code qui pourraient être évités.

A propos de la visibilité du package, il est intéressant d'attirer l'attention sur un cas particulier qui concerne les classes internes (« *inner classes* », déclarées à l'intérieur d'une autre classe). Il faut savoir que dans ce cas le compilateur Java enregistre tout de même la classe interne dans un fichier classe séparé. La classe interne, que l'on pourrait naïvement croire cachée à l'intérieur de sa classe englobante, devient ainsi aussi accessible qu'une autre. De plus, pour que la classe interne conserve la visibilité sur les variables d'instance de la classe englobante, le compilateur change automatiquement leur visibilité en *package*, même si elles ont été explicitement déclarées *privates* !

Dans la même optique de limitation des possibilités de réutilisation abusive des classes, on peut également rendre les classes non clonables (en Java, un clone est une nouvelle instance de classe créée par simple copie de l'image mémoire d'une autre instance). Cela évite de pouvoir créer une instance de classe sans en exécuter le constructeur. Pour éviter cela, il suffit de définir dans la classe une méthode *clone()* qui se contente de lancer une *java.lang.CloneNotSupportedException()*.

Un autre point sur la réutilisation des classes concerne la sérialisation. En effet, il peut être utile de rendre une classe non sérialisable lorsque cela n'est pas requis par l'application. N'oublions pas qu'un objet sérialisé peut être chargé en mémoire comme un simple tableau de bytes ; la structure interne de l'objet est par conséquent offerte à toute inspection ou analyse. Pour rendre un objet non sérialisable, il suffit de définir une méthode *writeObject()* qui lance simplement une exception. Cette méthode peut en outre être déclarée finale, afin d'éviter toute redéfinition.

Selon la même réflexion, et de façon plus subtile, il est possible également de rendre un objet non désérialisable ! La désérialisation est le mécanisme qui permet de transformer un flux d'octets en une instance de classe. On peut empêcher cette éventualité en déclarant une méthode *readObject()* qui lance une exception, et en la déclarant finale pour éviter toute redéfinition.

Un autre principe de sécurité purement liée au code offre un apparent paradoxe : l'usage du code signé lui-même est à éviter dans la mesure du possible ! Derrière cette curieuse contradiction se cache pourtant une évidence : lorsqu'on utilise la signature de code, c'est généralement pour pouvoir autoriser certaines opérations qui ne seraient pas autorisées pour du code non signé. Un code non signé n'a pas de privilège particulier ; et un code sans privilège particulier a moins la possibilité d'effectuer des opérations dommageables. La conclusion est d'utiliser la signature de code à bon escient, et non comme une solution de sécurité universelle.

De façon métaphorique, on pourrait dire qu'un mur disposant d'une porte bien gardée est pourtant moins sécurisé qu'un mur sans porte…

6.3. CONCLUSION

Tout au long du processus d'analyse que nous avons mis en oeuvre, nous pouvons distinguer plusieurs niveaux de réflexion :

- Les aspects liés à la programmation du langage orienté objet,
- Les aspects liés aux architectures d'applications distribuées,
- Les aspects de communication entre les composants,
- Les aspects de communications réseaux entre les localisations de code.

Selon les cas, la réflexion est menée selon trois axes essentiels :

- le point de vue de structure **microscopique** : aspects intervenants au niveau des objets ; ceci concerne essentiellement les diagrammes de classe et les objets signés/scellés/gardés.
- le point de vue de structure **macroscopique** : concerne la structure dans son ensemble, matérialisée par les liaisons entre les localisations dans les diagrammes de déploiement.
- le point de vue **globale** : qui concerne le système tout entier sans être spécifiquement lié à une structure particulière ; le point de vue global recouvre essentiellement l'administration des permissions (domaines, permissions spécifiques, gestion des permissions, gestion des sources de code, gestion des principaux, etc.).

Nous relèverons également que le **principe de moindre privilège** est une pierre angulaire de la sécurité quelle que soit la phase du processus de développement à laquelle on se situe.

Grâce à l'**intégration** des aspects de la sécurité dès le début du processus de développement, les vulnérabilités du système peuvent être mieux identifiées, ce qui permet d'envisager des contre-mesures plus efficaces.

En outre, cette approche autorise les développeurs qui ne sont pas experts en sécurité à intégrer les aspects de la sécurité dans un système. Cette remarque se justifie également au niveau implémentation : l'architecture de sécurité de Java propose en effet des composants standards couvrant la plupart des services de sécurité, et permet d'étendre ou de surcharger ces composants à l'aide du concept de fournisseurs de services (*CSP*).

L'idée maîtresse de ce guide est sans doute que la sécurité ne peut être vue comme un simple ajout à la modélisation d'un projet, mais constitue une véritable intégration dans tout le processus de conception d'un système. Loin d'être une étape supplémentaire ajoutée à la fin d'une méthode de développement, notre démarche traduit la nécessité d'un véritable ***tissage*** de la sécurité dans toutes les phases de développement d'un système.

PARTIE III :

EXEMPLE D'APPLICATION

Dans cette troisième partie, nous mettons en œuvre le Guide de sécurisation d'un projet Java qui concluait la seconde partie du mémoire.

Cette mise en œuvre se fait sur une application exemple dont nous décrivons le processus d'analyse complet, en y intégrant les notions de sécurité proposées dans le Guide.

7. APPLICATION : PNB

7.1. PERSONAL NET BANKING

Dans le cadre de ce mémoire, nous avons choisi une application de type bancaire, compte tenu de ses besoins évidents en matière de sécurité et de son contexte d'utilisation bien connu de tous. Ce type d'application permet de se passer d'une description détaillée pour mieux se focaliser sur les aspects d'analyse et de sécurité proprement dits.

Notre exemple d'application, que nous appellerons PNB (PERSONAL NET BANKING) est un simple prétexte à l'illustration des concepts et des techniques décrits dans les deux premières parties de ce mémoire. C'est pourquoi nous avons volontairement réduit les services disponibles afin d'éviter les développements superflus.

Nous n'avons pas cherché à fournir un exemple à tout prix réaliste du point de vue bancaire, mais bien de montrer comment mettre en œuvre certains aspects de la sécurité d'un projet orienté objet dans un environnent Java.

7.1.1. SERVICES

Le principal critère considéré pour le choix des services est de savoir si un point particulier apporte un éclairage supplémentaire sur un mécanisme de sécurité, qui ne serait pas encore mis en évidence par un autre point de l'application.

Dans le cadre des services disponibles, nous avons considéré deux types distincts : les services distants et les services internes à la banque.

Les **services distants** concernent les clients qui se connectent à distance sur le serveur bancaire, c'est-à-dire depuis un poste situé en dehors du réseau local de la banque :

- Consultation du solde du compte.
- Consultation de l'historique du compte (dernières opérations).
- Impression de l'historique du compte (« extraits de compte »).
- Virement depuis le compte courant.
- Changement du code d'accès.

Les **services internes** s'ajoutent aux services précédents, et concernent les activités des gestionnaires internes de la banque, qui accèdent à l'application à travers le réseau local :

- Consultation d'un compte.
- Modification des données signalétiques d'un titulaire de compte.
- Blocage d'un compte (restriction d'accès en lecture seulement).

7.1.2. CONTRAINTES

Toujours dans un but de simplification, nous avons ajouté les contraintes suivantes :

- Un client dispose d'un et un seul compte.
- Tous les comptes considérés sont dans la même banque (p.ex. pour le cas des virements).
- Les comptes sont considérés comme existants et approvisionnés (PNB ne gère pas la création d'un nouveau client ni la création d'un nouveau compte).
- Un client est supposé avoir obtenu préalablement son nom d'utilisateur et son mot de passe (PNB ne gère pas la distribution des données de connexion).
- Un compte client bloqué le reste indéfiniment (PNB ne gère pas le déblocage d'un compte).
- Une connexion établie avec le serveur reste opérationnelle (PNB ne gère pas la rupture accidentelle de la communication avec le système).
- Un gestionnaire interne ne peut pas changer son propre mot de passe, ni le mot de passe d'un client de la banque.

7.1.3. DÉMARCHE

Nous allons suivre la méthode de développement que nous avons décrite dans la deuxième partie du mémoire. Nous appliquons la modélisation UML telle que présentée dans le chapitre 5, en y intégrant le Guide de sécurisation du chapitre 6.

Soulignons que certains des aspects présentés dans le Guide ne sont pas repris dans PNB ; c'est une des raisons pour lesquelles nous avons voulu présenter cette application séparément.

> **NOTE** : par manque de temps, il ne nous a pas été possible de sécuriser complètement l'application exemple. Nous nous sommes donc limité à sécuriser quelques parties représentatives des idées développées dans le Guide. Pour certaines parties, nous donnons simplement un début de réflexion basées sur le Guide, sans développement approfondi.
> Soulignons toutefois que l'analyse UML présentée ici, qui reste pour partie non sécurisée, est le résultat d'une réflexion sur les moyens de mettre en évidence les idées du Guide. Une version même non sécurisée représente donc déjà un intérêt en soit, comme terrain propice à l'exploration du guide de sécurisation.

7.2. EXPRESSION DES BESOINS

L'univers bancaire comporte des besoins de confidentialité et d'intégrité évidents pour toutes les ressources touchant au cœur même de son activité.

Une partie d'entre eux concernent des ressources physiques, essentiellement réseau, qu'il faudrait évidemment prendre en compte dans le cadre d'un projet réel. Ceci inclut d'autres types de services tels que la disponibilité de service.

D'un point de vue informations, ces besoins concernent des ressources telles que les comptes, les opérations sur les comptes, l'identification des utilisateurs, ou les codes d'accès des utilisateurs. Ils touchent donc aux services de vérification d'identité, de preuve de non répudiation et de permissions d'accès. Un contrôle actif peut également être mis en œuvre avec un service d'audit.

Tous ces besoins mériteraient une étude de risque approfondie, incluant notamment une évaluation des coûts liés aux pertes potentielles consécutives à l'exploitation d'une vulnérabilité, et les coûts liés à la mise en œuvre de contre-mesures destinées à réduire cette vulnérabilité. La politique de sécurité serait alors définie en fonction du niveau de risque acceptable obtenu.

7.3. SCHÉMA CONCEPTUEL

Le diagramme de classe ci-dessous est une vue conceptuelle du système PNB ; elle permet de fixer les éléments du domaine étudié et leurs relations. Ce diagramme de classe est le pendant fonctionnel du diagramme de classe orienté objets de la phase d'analyse (voir *5.1.3. Analyse*).

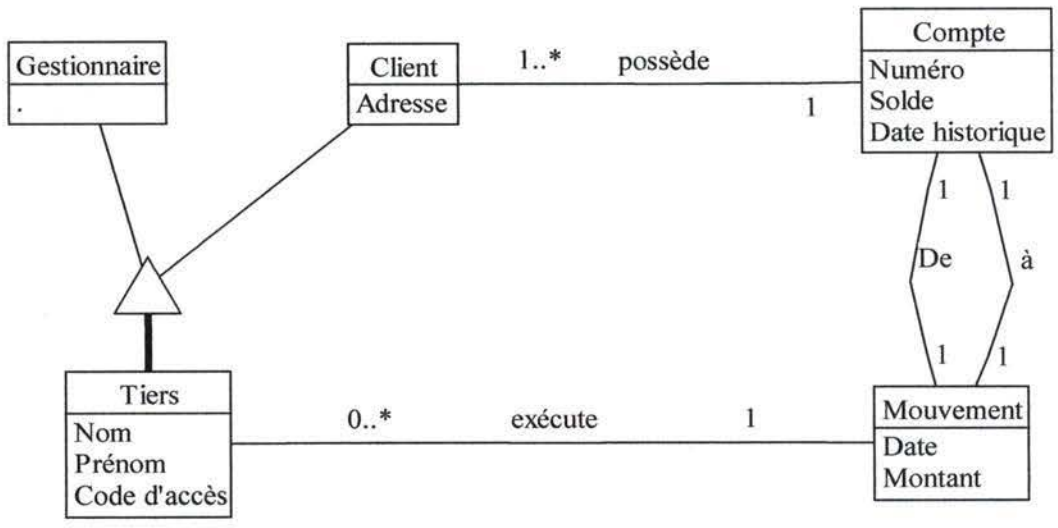


FIG. 58 – PNB : SCHÉMA CONCEPTUEL

7.4. USE CASES

7.4.1. DIAGRAMME DE USE CASES

Les uses cases de l'application PNB sont au nombre de dix : Connexion au système, Choix du compte, Solde du compte, Historique du compte, Impression de l'historique, Modification signalétique, Blocage du compte, Changement du code d'accès, Déconnexion du système.

On identifiera deux acteurs pour ces use cases : le client (utilisateur externe) et le gestionnaire (utilisateur interne). Les relations avec les acteurs, et les relations entre use cases sont représentées dans le **diagramme de use cases** ci-dessous.

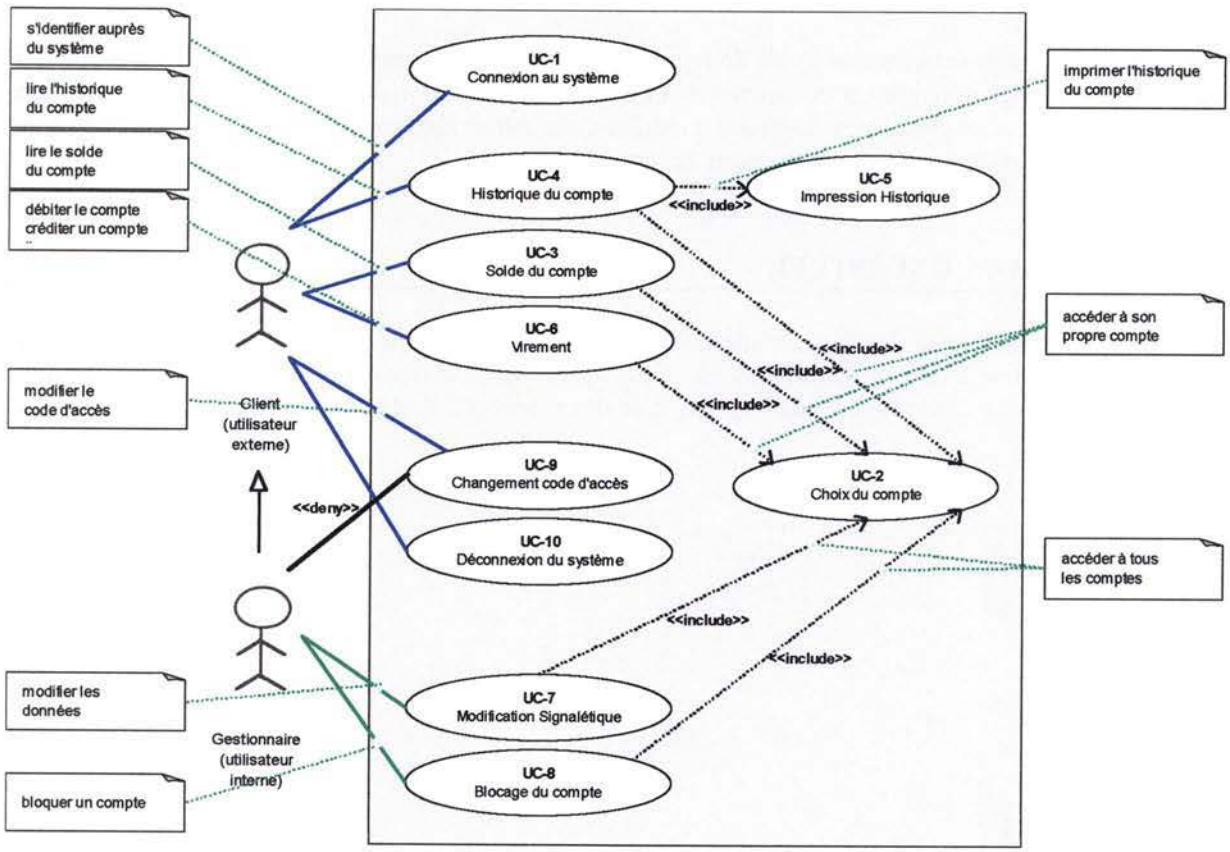


FIG. 59 – PNB : DIAGRAMME DE USE CASES

Nous remarquerons sur le diagramme de use cases que le gestionnaire interne hérite des actions disponibles pour le client externe. Il hérite également des autorisations d'accès associées, sauf en ce qui concerne le droit de modifier le mot de passe lié à un compte, ce qui est traduit par une spécification *deny*.

7.4.2. HIÉRARCHIE DE USE CASES

Une autre représentation des use cases et de leurs relations peut être donnée sous la forme d'une **hiérarchie de use cases**. Nous avons distingué ici les use cases des utilisateurs externes et internes par une coloration différente.

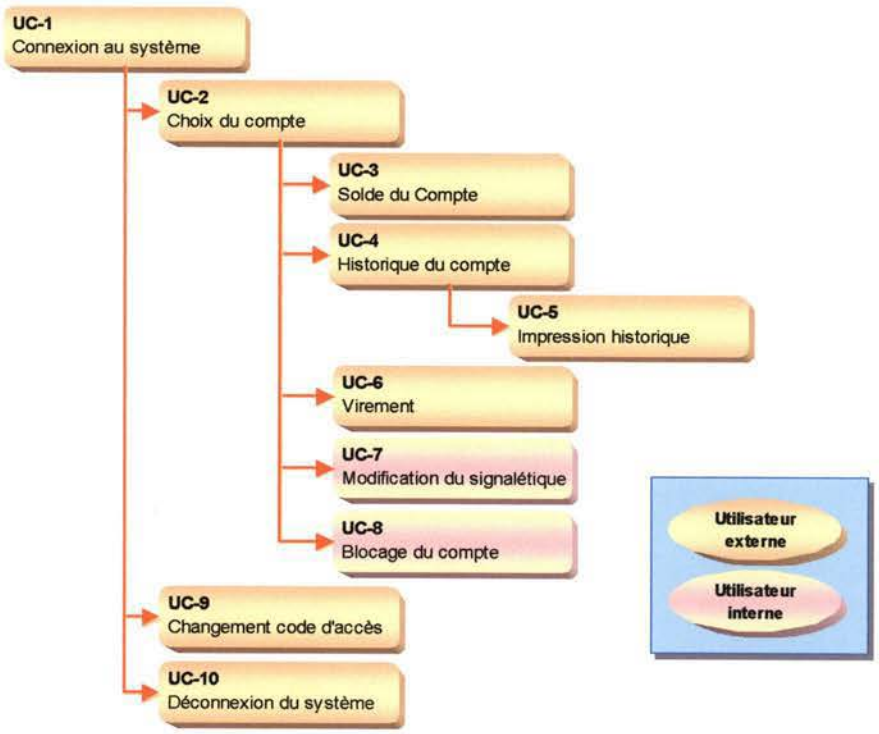


FIG. 60 – PNB : HIÉRARCHIE DE USE CASES

Cette hiérarchie peut être lue comme la description des *pré-conditions* pour chaque use case, subordonnant le déclenchement de chacun à la réalisation préalable d'un autre use case (sauf pour UC♦1 qui ne nécessite aucune pré-condition du système, s'agissant de la connexion elle-même).

7.4.3. USE CASE 1 : CONNEXION AU SYSTÈME

DESCRIPTION :
UC-1 gère la connexion au système, qui est un préalable à tout autre use case.

ETAPES :
1. Le client démarre l'application (grâce à un son « module » client).
2. L'application se connecte au serveur d'application.
3. Le client s'identifie auprès du serveur :
   - Par un nom d'utilisateur.
   - Par un mot de passe (ou « pincode »).
4. Le système détermine le type de client (Interne ou Externe).
5. Le système propose les options disponibles en fonction du type de client :
   - Externe : voir UC-2 à UC-6, plus UC-9.
   - Interne : voir UC-2 à UC-8.

PRÉCONDITIONS :

- L'utilisateur est déjà connu de la banque.
- Il possède son nom d'utilisateur (fourni par la banque) et son mot de passe (l'application PNB ne gère pas l'inscription d'un nouveau client ni la distribution des données d'accès).

CAS D'EXCEPTIONS :

- Le serveur n'est pas accessible.
- Le nom d'utilisateur est inconnu.
- Le mot de passe est incorrect.

SEQUENCE DIAGRAMS :

Cas Normal

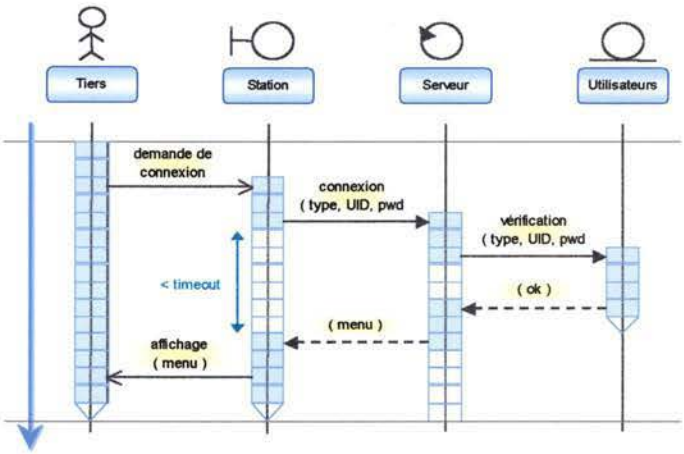


FIG. 61 – PNB : SEQUENCE DIAGRAM UC1 : CAS NORMAL

Cas d'Exception 1 : Délai de connexion dépassé

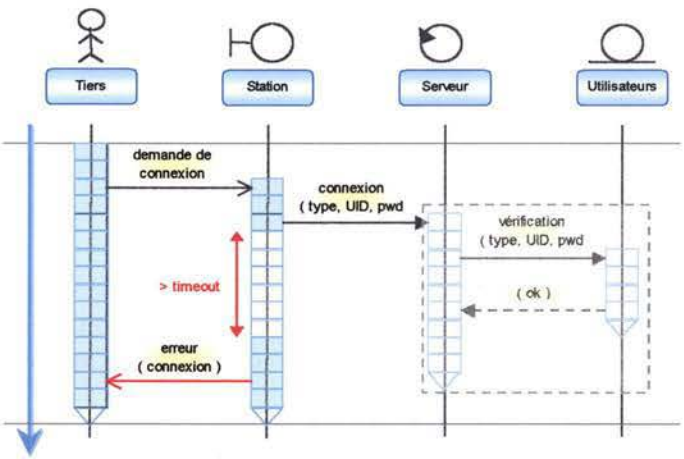


FIG. 62 – PNB : SEQUENCE DIAGRAM UC1 : CAS D'EXCEPTION 1

Cas d'Exception 2 : Erreur UID / pwd

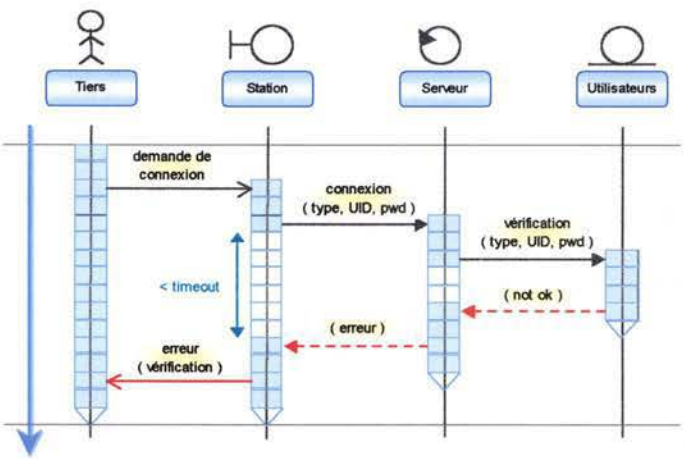


FIG. 63 – PNB : SEQUENCE DIAGRAM UC1 : CAS D'EXCEPTION 2

7.4.4. USE CASE 2 : CHOIX DU COMPTE

DESCRIPTION :
UC-2 propose la sélection du compte sur lequel l'utilisateur va travailler. Ce use case peut être scindé en deux utilisations distinctes, selon que l'utilisateur est externe ou interne au système :

UC-2A : Utilisateur externe.
Selon les contraintes de départ, son numéro de compte est unique et donc connu automatiquement du système (il ne peut avoir accès qu'à son propre compte). UC-2 est donc une opération transparente pour un utilisateur externe.

UC-2B : Utilisateur interne.
Un utilisateur interne (gestionnaire de la banque) a accès à tous les comptes de la banque. UC-2B sert donc à identifier le compte sur lequel l'utilisateur interne veut travailler.

ETAPES :
1. L'utilisateur introduit le numéro de compte (automatique si UC-2A).
2. Le système valide le numéro de compte (implicite si UC-2A).

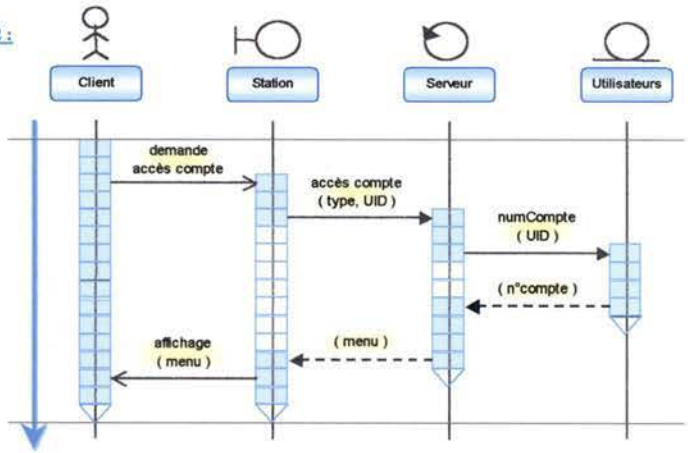
PRÉCONDITIONS :
- UC-1.

CAS D'EXCEPTIONS :
- Le numéro de compte est invalide.
- Le numéro de compte est inexistant.
- Le compte est bloqué.

SEQUENCE DIAGRAMS :

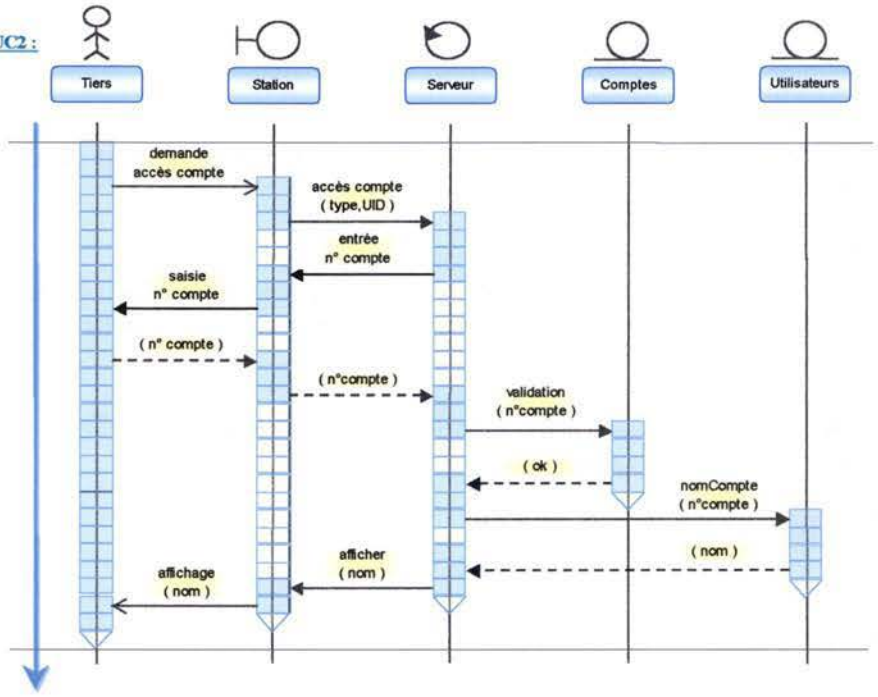
Cas Normal A :
Compte de l'utilisateur externe

FIG. 64 – PNB : SEQUENCE DIAGRAM UC2 : CAS NORMAL A



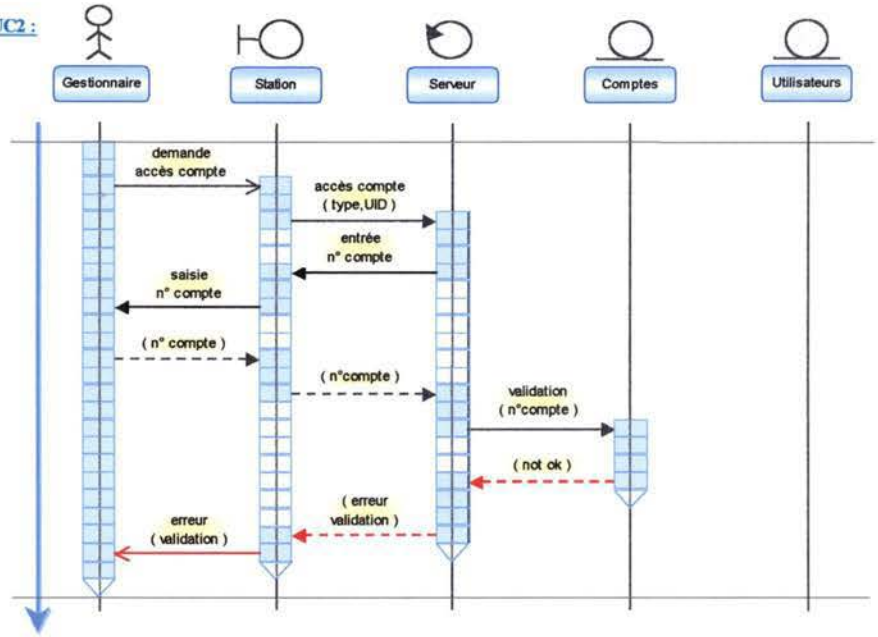
Cas Normal B :
Compte d'un tiers

FIG. 65 – PNB : SEQUENCE DIAGRAM UC2 : CAS NORMAL B



Cas d'Exception B1 :
Numéro de compte incorrect ou inexistant

FIG. 66 – PNB : SEQUENCE DIAGRAM UC2 : CAS D'EXCEPTION B1



7.4.5. USE CASE 3 : SOLDE DU COMPTE

DESCRIPTION :
UC-3 permet d'afficher le solde du compte de l'utilisateur.

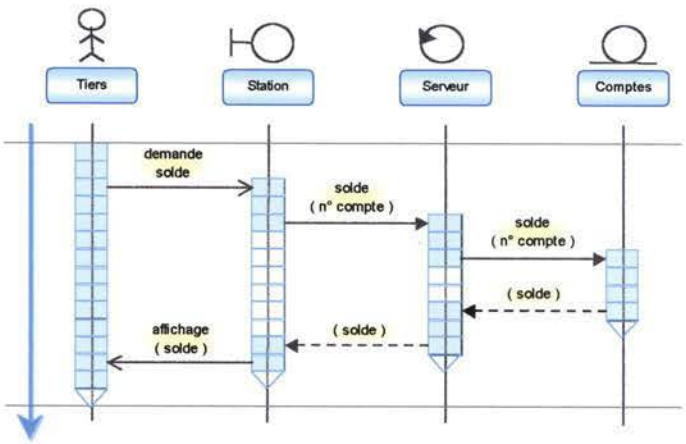
ETAPES :
1. Le système affiche le solde du compte.

PRÉCONDITIONS :
- UC-2.

CAS D'EXCEPTIONS :
- /

SEQUENCE DIAGRAMS :

FIG. 67 – PNB : SEQUENCE DIAGRAM UC3



7.4.6. USE CASE 4 : HISTORIQUE DU COMPTE

DESCRIPTION :
UC-4 permet d'afficher l'historique des opérations sur le compte ; l'historique reprend les opérations effectuées depuis la dernière impression, « point de départ » de l'historique (voir UC-5).

ETAPES :
1. Le système affiche la liste des opérations depuis la dernière impression.

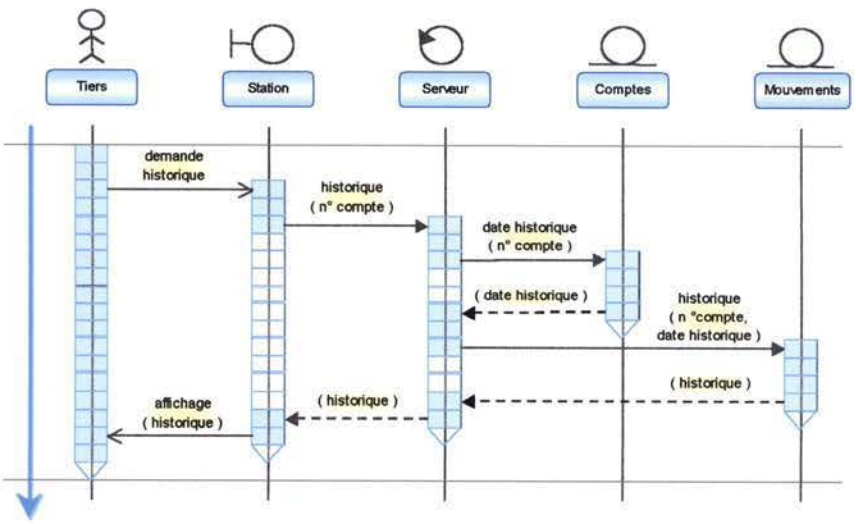
PRÉCONDITIONS :
- UC-2.

CAS D'EXCEPTIONS :
- La liste des opérations depuis le «point de départ » de l'historique est vide.

SEQUENCE DIAGRAMS :

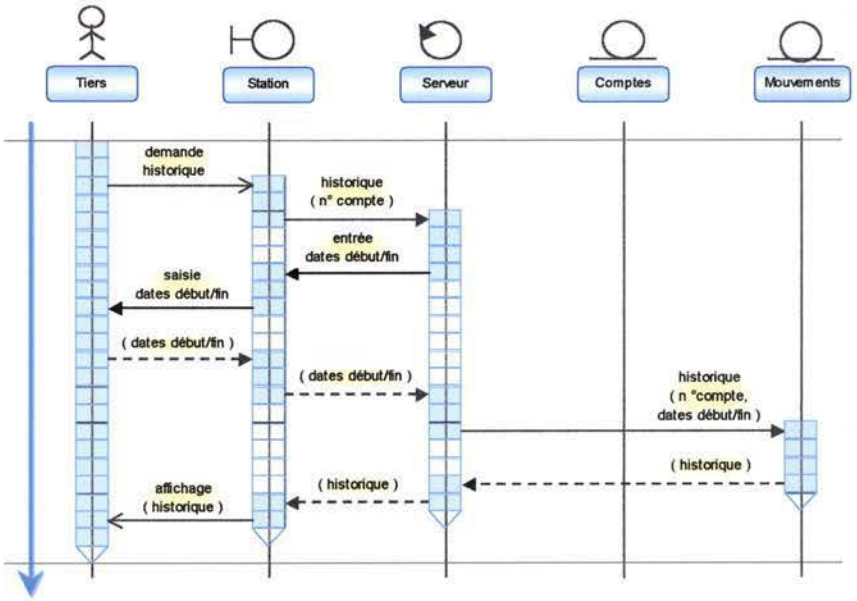
Cas Normal A : Utilisateur Externe

FIG. 68 – PNB : SEQUENCE DIAGRAM UC4 : CAS NORMAL A



Cas Normal B :
Utilisateur Interne

FIG. 69 – PNB : SEQUENCE DIAGRAM UC4 : CAS NORMAL B



7.4.7. USE CASE 5 : IMPRESSION HISTORIQUE

DESCRIPTION :
UC-5 permet d'imprimer l'historique des opérations sur le compte affiché par UC-4 ; l'historique reprend les opérations effectuées depuis la dernière impression (« point de départ » de l'historique). Si la liste des opérations depuis le «point de départ » de l'historique est vide, on considère que l'impression produit une liste notifiant l'absence d'historique. L'impression a donc toujours lieu.

ETAPES :
1. Le système imprime la liste des opérations depuis la dernière impression (même vide).
2. Le système réinitialise le point de départ de l'historique (pour prochain UC-4 ou UC-5).

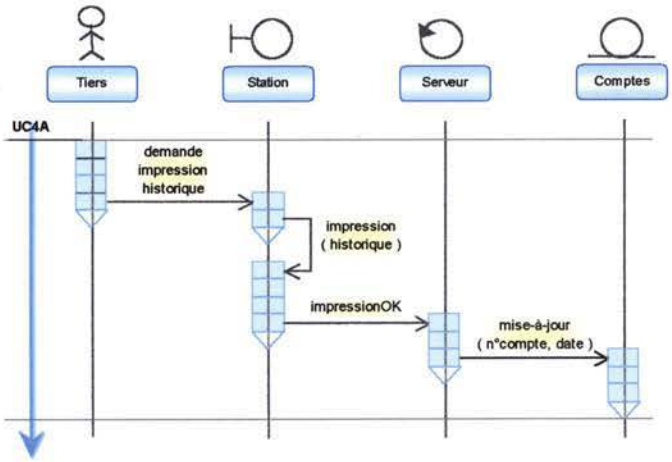
PRÉCONDITIONS :
- UC-4.

CAS D'EXCEPTIONS :
- Problème d'impression (sur la station de travail).

SEQUENCE DIAGRAMS :

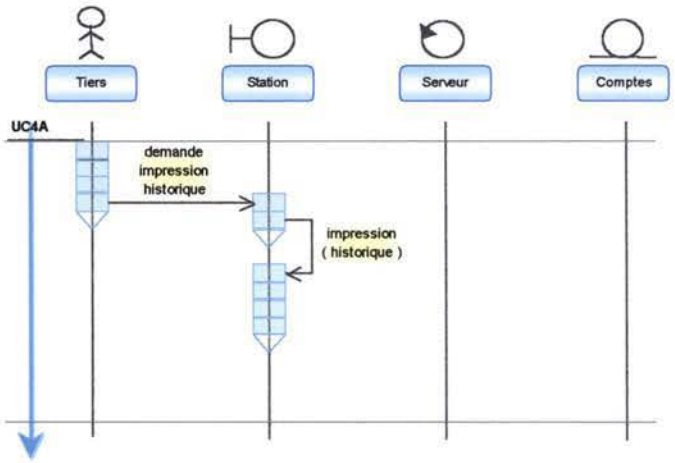
Cas Normal A :
Utilisateur Externe

FIG. 70 – PNB : SEQUENCE DIAGRAM UC5 : CAS NORMAL A



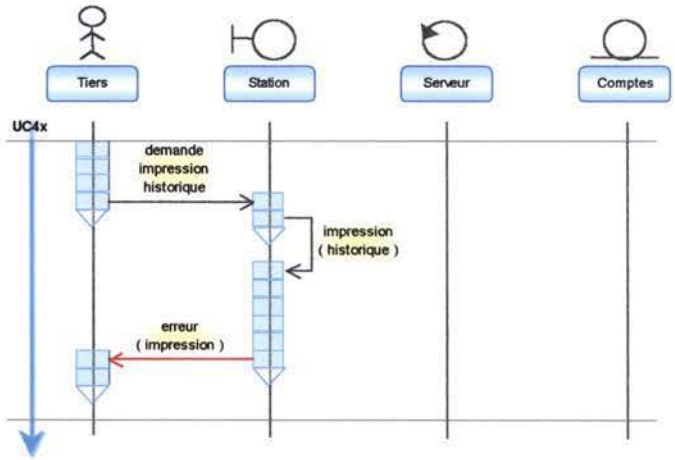
Cas Normal B :
Utilisateur interne

FIG. 71 – PNB : SEQUENCE DIAGRAM UC5 : CAS NORMAL B



Cas D'exception :
Erreur d'Impression

FIG. 72 – PNB : SEQUENCE DIAGRAM UC5 : CAS D'EXCEPTION



7.4.8. USE CASE 6 : VIREMENT

DESCRIPTION :
UC-6 permet à d'exécuter un virement au départ du compte de l'utilisateur, vers un autre compte de la banque (voir contraintes de départ).

ETAPES :
1. L'utilisateur introduit le numéro du compte cible.
2. L'utilisateur introduit le montant (positif) du virement.
3. Le système date l'opération.
4. Le système valide l'opération.
5. Le système exécute l'opération.

PRÉCONDITIONS :
- UC-2.
- Le compte n'est pas bloqué.

CAS D'EXCEPTIONS :
- Le compte cible est invalide.
- Le compte cible est inconnu.
- Le compte n'est pas suffisamment approvisionné.

SEQUENCE DIAGRAMS :

Cas Normal

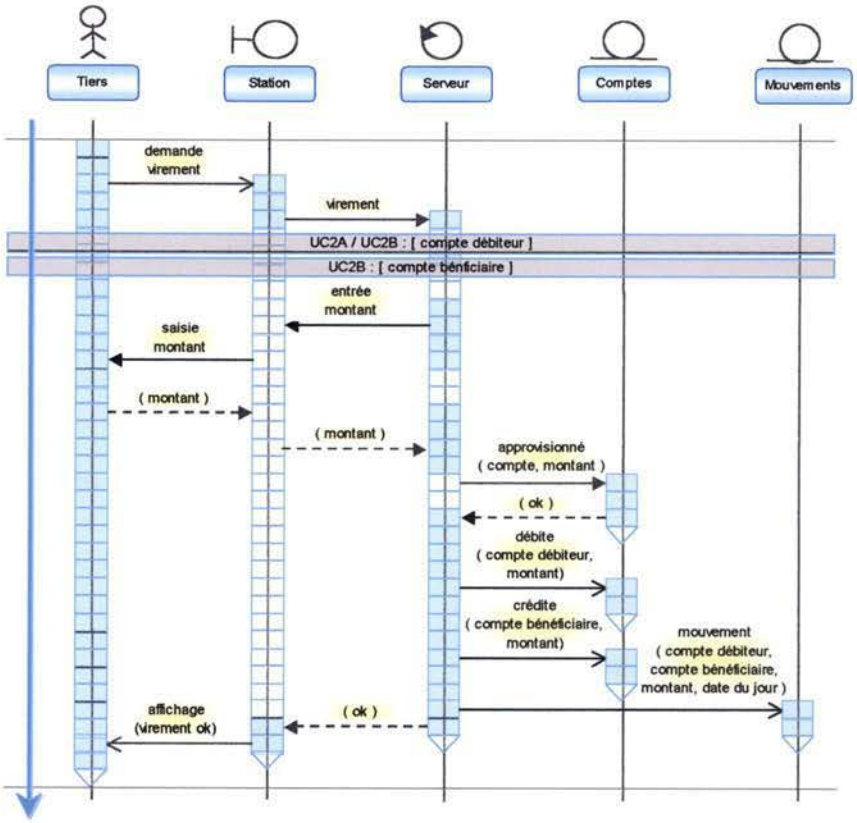


FIG. 73 – PNB : SEQUENCE DIAGRAM UC6 : CAS NORMAL

NOTE : l'opération de virement en elle-même constitue une transaction au sens base de données : les opérations débit, crédit et mouvements constituent un ensemble atomique (elles ont lieu toutes les trois ou pas du tout). De ce point de vue, le mécanisme est assez proche d'un protocole de sécurité, si l'on considère qu'une attaque possible consisterait à intervenir au milieu de la transaction (par exemple en permettant une action de débit/crédit sans contrepartie équilibrée). Voir *6.2.3. Analyse orientée objets*.

Cas d'Exception : Compte non Approvisionné

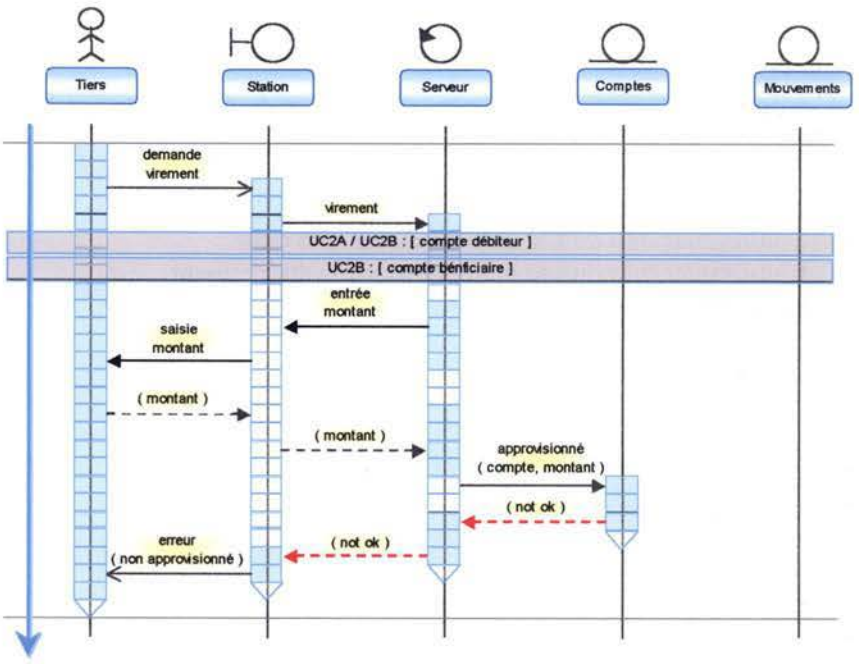


FIG. 74 – PNB : SEQUENCE DIAGRAM UC6 : CAS D'EXCEPTION

7.4.9. USE CASE 7 : MODIFICATION DU SIGNALÉTIQUE (INTERNE)

DESCRIPTION :
UC-7 permet à un gestionnaire interne de modifier les données du titulaire du compte. Ces données ont un caractère purement signalétique (nom, adresse,...) et ne concernent pas le compte lui-même.

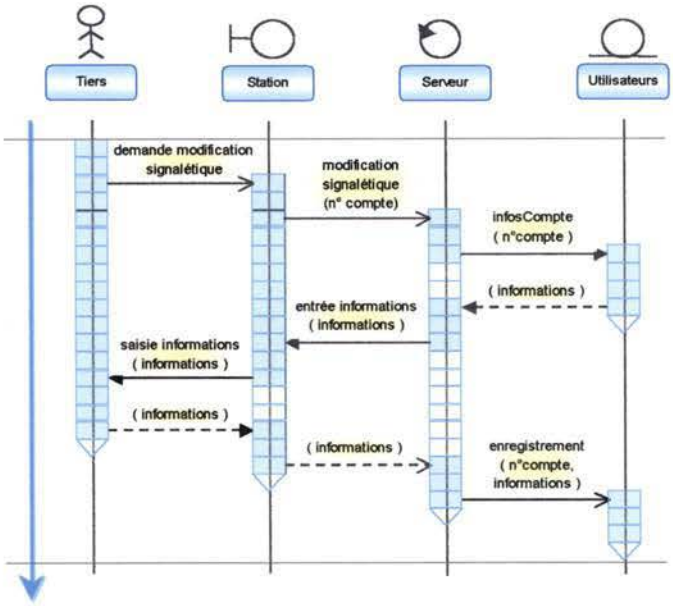
ETAPES :
1. L'utilisateur introduit les modifications des données signalétiques et valide sa saisie.
2. Le système prend en compte les modifications opérées par l'utilisateur.

PRÉCONDITIONS :
- UC-2.
- L'utilisateur est un gestionnaire interne.
- Le compte n'est pas bloqué.

CAS D'EXCEPTIONS :
- /

SEQUENCE DIAGRAMS :

FIG. 75 – PNB : SEQUENCE DIAGRAM UC7



7.4.10. USE CASE 8 : BLOCAGE DU COMPTE (INTERNE)

DESCRIPTION :
UC-8 permet à un gestionnaire interne de bloquer un compte. Un compte bloqué n'est plus accessible que pour lecture seule par tout utilisateur (UC-6, UC-7 et UC-8 sont interdits).
L'application ne prévoit pas de pouvoir débloquer un compte bloqué (voir contraintes de départ).

ETAPES :
1. L'utilisateur confirme le blocage du compte.
2. Le système prend en compte les modifications opérées par l'utilisateur.

PRÉCONDITIONS :
- UC-2.
- L'utilisateur est un gestionnaire interne.
- Le compte n'est pas déjà bloqué.

CAS D'EXCEPTIONS :
- /

SEQUENCE DIAGRAMS :

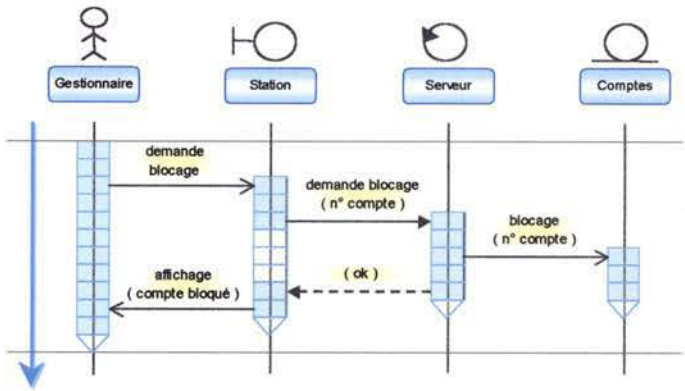


FIG. 76 – PNB : SEQUENCE DIAGRAM UC8

7.4.11. USE CASE 9 : CHANGEMENT DU CODE D'ACCÈS

DESCRIPTION :
UC-9 permet à le client connecté au système de modifier son mot de passe. Un client ne peut modifier que son propre mot de passe.
On considère que la technique du double encodage est suffisante pour valider le nouveau code d'accès non vide ; la demande est validée par une vérification du mot de passe actuel.

ETAPES :
1. L'utilisateur introduit le nouveau code d'accès non vide (mot de passe) et valide sa saisie.
2. Le système prend en compte le nouveau code d'accès pour ce client.

PRÉCONDITIONS :
- UC-1.
- L'utilisateur est un client.

CAS D'EXCEPTIONS :
- Le code d'accès actuel est incorrect.

SEQUENCE DIAGRAMS :

Cas Normal

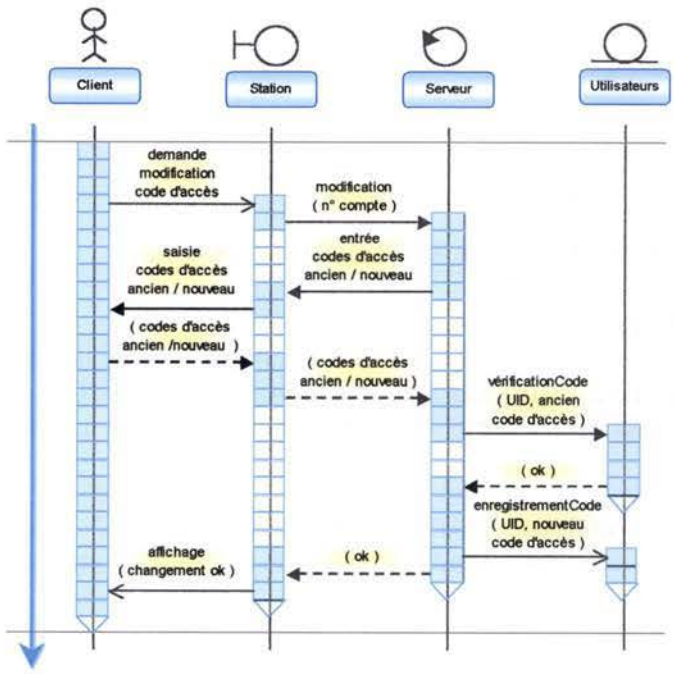


FIG. 77 – PNB : SEQUENCE DIAGRAM UC9 : CAS NORMAL

Cas d'Exception :
Code d'accès actuel incorrect

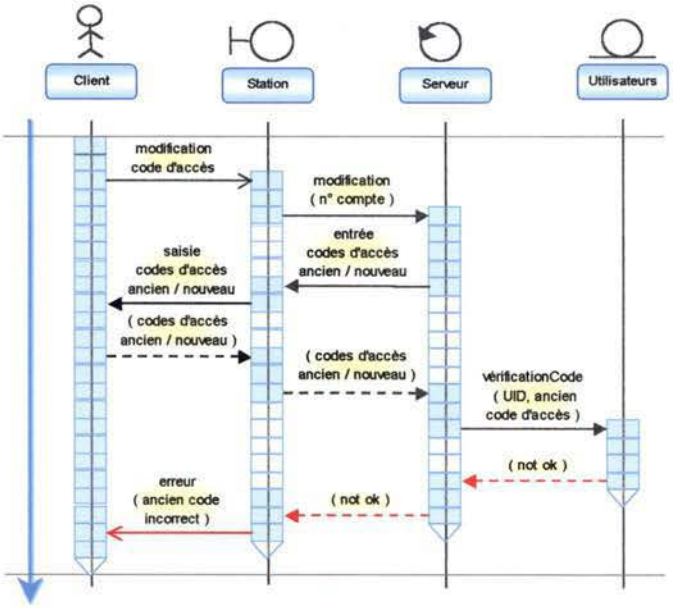


FIG. 78 – PNB : SEQUENCE DIAGRAM UC9 : CAS D'EXCEPTION

7.4.12. USE CASE 10 : DÉCONNEXION DU SYSTÈME

DESCRIPTION :
UC-10 permet de se déconnecter du système (fin de session). La déconnexion a lieu soit à la demande de l'utilisateur, soit de façon automatique à l'expiration d'un *timer* d'inactivité sur la session.

ETAPES :
1. La déconnexion du système est requise (par l'utilisateur ou via timeout).
2. Le système déconnecte l'utilisateur et libère les ressources allouées.

PRÉCONDITIONS :
- UC-1.

CAS D'EXCEPTIONS :
- /

SEQUENCE DIAGRAMS :

Cas Normal A :
Déconnexion Par l'Utilisateur

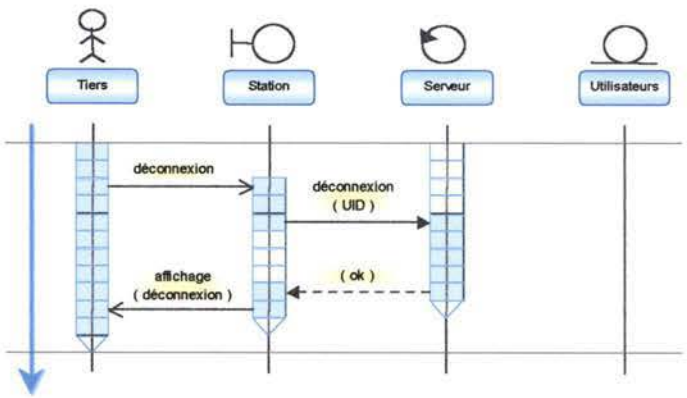


FIG. 79 – PNB : SEQUENCE DIAGRAM UC10 : CAS NORMAL A

Cas Normal B :
Déconnexion Automatique

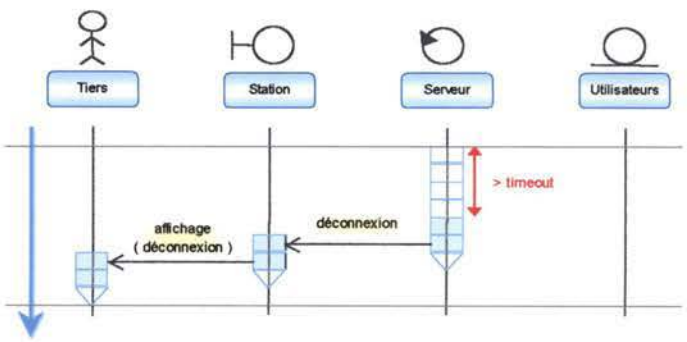


FIG. 80 – PNB : SEQUENCE DIAGRAM UC10 : CAS NORMAL B

7.5. DESCRIPTION DES COMPOSANTS ACTIFS

FIG. 81 – TABLEAU : PNB : DESCRIPTION DES COMPOSANTS ACTIFS - STATION

STATION	STATION (version agrégée)
Demande Connexion()	**Connexion(**)
Affichage(IN Menu)	**Affichage (** IN *{ Menu,*
Erreur(IN Connexion)	*Nom,*
Erreur(IN Vérification)	*Solde,*
Demande accès compte()	*Historique,*
Entrée numCompte(OUT n°compte)	*Virement ok,*
Saisie numCompte(OUT n°compte)	*Compte bloqué,*
Envoie(OUT n°compte)	*Changement ok,*
Affichage(IN nom)	*Déconnexion }*)
Erreur(IN validation)	**Entrée(** INOUT *{ N°Compte,*
Demande Solde	*Date début / fin,*
Affichage(IN solde)	*Montant,*
Demande Historique	*Code d'accès ,*
Affichage(IN historique)	*ancien/nouveau }*)
Entrée dates début/fin(OUT : dates début/fin)	**Demande(** IN *{ Accès Compte,*
Saisie dates début/fin(OUT : dates début/fin)	*Solde,*
Demande Impression Historique	*Historique,*
Impression(IN historique)	*Impression Historique,*
Demande virement	*Virement,*
Entrée montant(OUT montant)	*Modification Signalétique,*
Saisie montant(OUT montant)	*Blocage,*
Affichage(IN virement ok)	*Modification code d'accès,*
Erreur(IN non approvisionné)	*Déconnexion }*)
Demande modification signalétique	**Saisie(** INOUT *{ N°Compte,*
Saisie informations(INOUT informations)	*Dates début/fin,*
Entrée informations(INOUT informations)	*Montant,*
Demande blocage	*Informations,*
Affichage(IN compte bloqué)	*Codes d'accès*
Demande modification code d'accès	*ancien/nouveau }*)
Entrée codes d'accès ancien/nouveau (OUT codes d'accès ancien/nouveau)	**Impression(** IN *Historique*)
Saisie codes d'accès ancien/nouveau (OUT codes d'accès ancien/nouveau)	
Affichage(IN changement ok)	
Erreur(IN ancien code incorrect)	
Demande de déconnexion	
Affichage(IN déconnexion)	

FIG. 82 – TABLEAU : PNB : DESCRIPTION DES COMPOSANTS ACTIFS - SERVEUR

SERVEUR	SERVEUR (version agrégée)
Connexion(IN type, UID, pwd, OUT menu)	**Connexion(** IN *Type {client,gestionnaire},*
Connexion(IN type, UID, pwd OUT erreur)	*UID,*
Accès Compte(IN type,UID, OUT menu)	*Pwd,*
Validation(IN n°compte, OUT erreur validation)	OUT *Menu,*
Solde(IN n°compte, OUT solde)	*Erreur* **)**
Historique(IN n°compte, OUT historique)	**AccèsCompte(** IN *Type {client,gestionnaire},*
ImpressionOk()	*UID,*
Virement(OUT ok)	OUT *Menu* **)**
Demande blocage (IN n°compte, OUT ok)	**Validation(** IN *N°Compte,*
Modification(IN n°compte, OUT ok)	OUT *Nom,*
Déconnexion(IN UID, OUT ok)	*Erreur* **)**
	Solde(IN *N°Compte,*
	OUT *Solde* **)**
	Historique(IN *N°Compte,*
	OUT *Historique* **)**
	ImpressionOk(**)**
	Virement(OUT *Ok* **)**
	DemandeBlocage(IN *N°Compte,*
	OUT *Ok* **)**
	Modification(IN *N°Compte,*
	OUT *Ok)*
	Déconnexion(IN *UID,*
	OUT *Ok* **)**

FIG. 83 – TABLEAU : PNB : DESCRIPTION DES COMPOSANTS ACTIFS – BD.TIERS

BD.TIERS	BD.TIERS (version agrégée)
Vérification(IN type,UID,pwd, OUT ok)	**Vérification(** IN *Type,*
NumCompte(IN UID, OUT n°compte)	*UID,*
NomCompte(IN n°compte, OUT nom)	*Pwd*
InfosCompte (IN n°compte, OUT informations)	OUT *Ok* **)**
Enregistrement (IN n°compte, informations)	**NumCompte(** IN *UID,*
Vérification Code (IN UID, ancien code, OUT ok)	OUT *N°Compte* **)**
Enregistrement Code(IN UID, nouveau code)	**NomCompte(** IN *UID,*
	OUT *Nom* **)**
	InfosCompte(IN *N°Compte,*
	OUT *Informations* **)**
	Enregistrement(IN *N°Compte,*
	Informations **)**
	EnregistrementCode(IN *UID,*
	Nouveau Code **)**

FIG. 84 – TABLEAU : PNB : DESCRIPTION DES COMPOSANTS ACTIFS – BD.COMPTES

BD.COMPTES	BD.COMPTES (version agrégée)
Validation(IN n°compte, OUT ok)	**Validation(** IN *N°Compte*
Solde (IN n°compte, OUT solde)	OUT *Ok*)
Date Historique (IN n°compte, OUT date)	**Solde(** IN *N°Compte*
Mise à jour(IN n°compte, date)	OUT *Solde*)
Approvisionné(IN n°compte, montant, OUT ok)	**DateHistorique(** IN *N°compte*
Débite(IN n°compte débiteur, montant)	OUT *Date*)
Crédite(IN n°compte bénéficiaire, montant)	**MiseAJour (** IN *N°Compte*
Blocage(IN n°compte)	*Date*)
	Approvisionné(IN *N°Compte*
	Montant
	OUT *Ok*)
	Opération(IN *N°Compte*
	Type {débit/crédit}
	Montant)
	Blocage(IN *N°Compte*)

FIG. 85 – TABLEAU : PNB : DESCRIPTION DES COMPOSANTS ACTIFS – BD.MOUVEMENTS

BD.MOUVEMENTS	BD.MOUVEMENTS (version agrégée)
Historique (IN n°compte, date historique, OUT historique)	**Historique(** IN *N°Compte,*
	DateDe {début, historique},
Historique(IN n°compte, date début/fin, OUT historique)	*DateA {fin, du jour},*
	OUT *Historique)*
Mouvement(in compte débiteur,	**Mouvement(** IN *Débiteur,*
compte bénéficiaire,	*Bénéficiaire,*
montant, date du jour)	*Montant,*
	Date)

7.6. DIAGRAMME DE CLASSE

Le diagramme de classe construit sur base des composants actifs est donné ci-dessous. Afin d'éviter de surcharger inutilement le graphique, nous n'avons pas représenté les arguments des méthodes associées aux différentes classes.

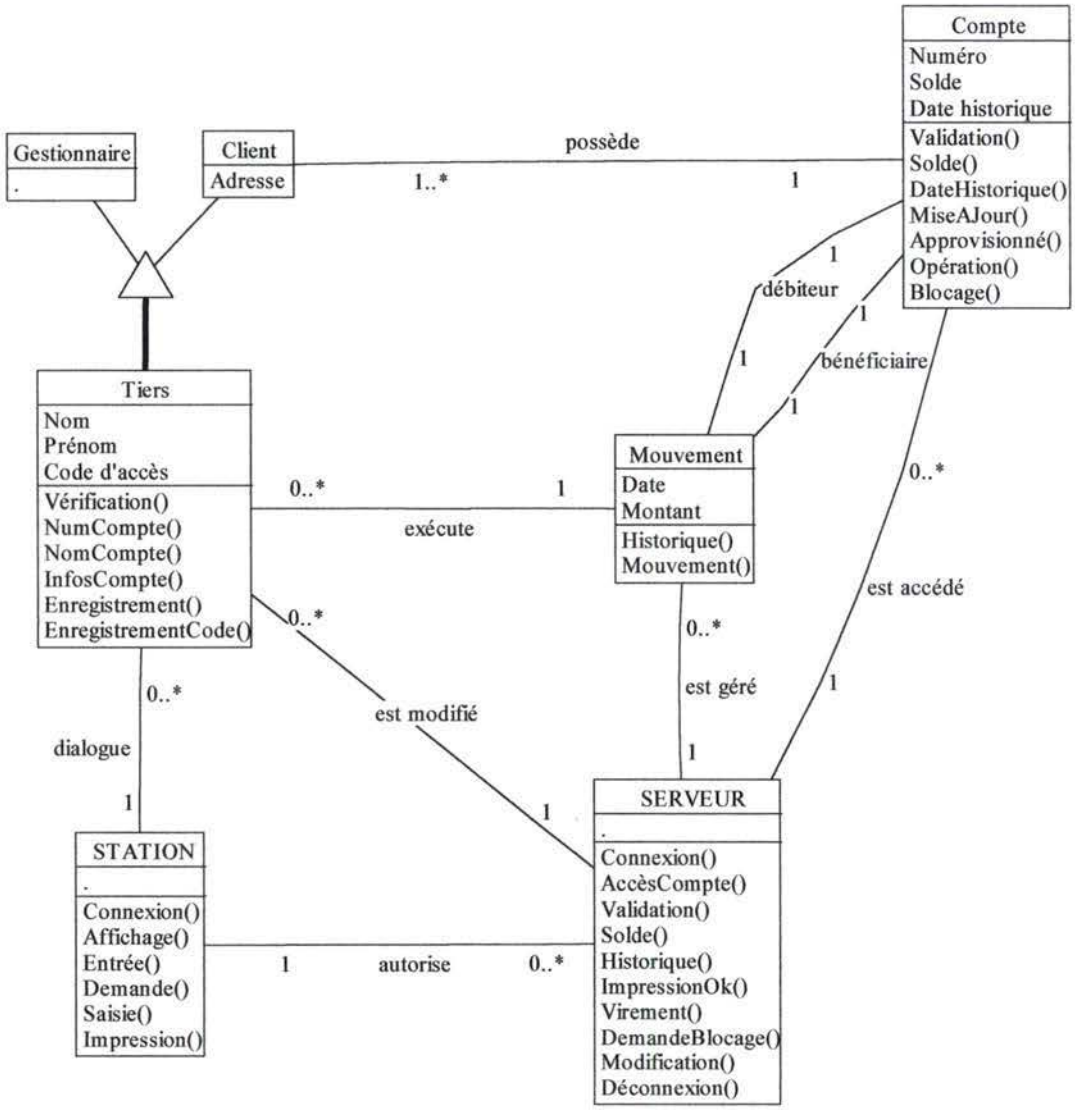


FIG. 86 – PNB : DIAGRAMME DE CLASSE

7.7. ARCHITECTURE LOGIQUE DU SYSTÈME

Le schéma d'architecture logique du système de figure pas à proprement parler dans notre démarche de développement orienté objets. Toutefois, comme nous l'avons mentionné dans la partie analyse, il constitue un complément utile au diagramme de déploiement.

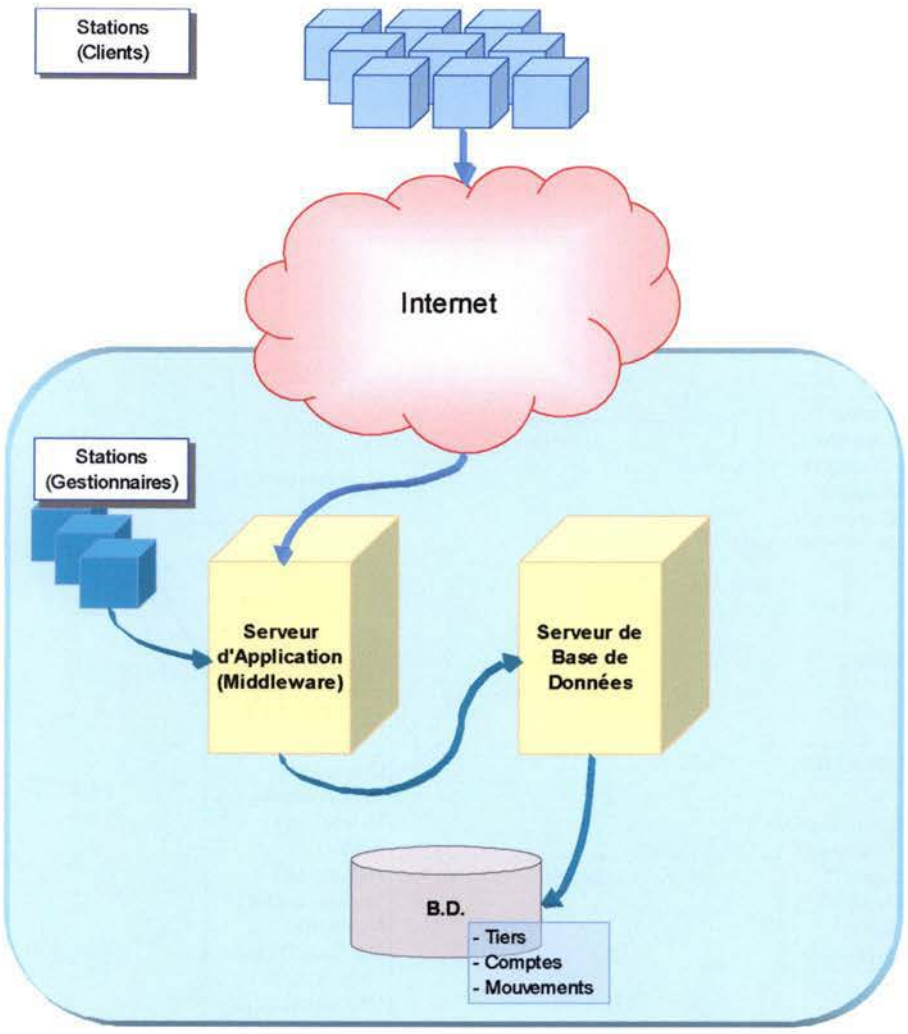


FIG. 87 – PNB : ARCHITECTURE LOGIQUE DU SYSTÈME

NOTE : il est également envisageable d'implémenter la base de données sous une forme distribuée, ce qui permettrait de rencontrer plus efficacement le service de sécurité de disponibilité de service. Le choix de ce type d'implémentation s'effectue au niveau de l'architecture logique, afin de permettre une définition plus aisée des localisations au niveau physique lors du déploiement.

7.8. DIAGRAMME DE DÉPLOIEMENT DES COMPOSANTS

Comme indiqué sur le diagramme de déploiement, le protocole de communication utilisé pour les liaisons physiques est TCP/IP. Les liaisons logiques entre les composants sont basées sur la technologie de communication RMI. La sécurisation des communications est assurée par SSL.

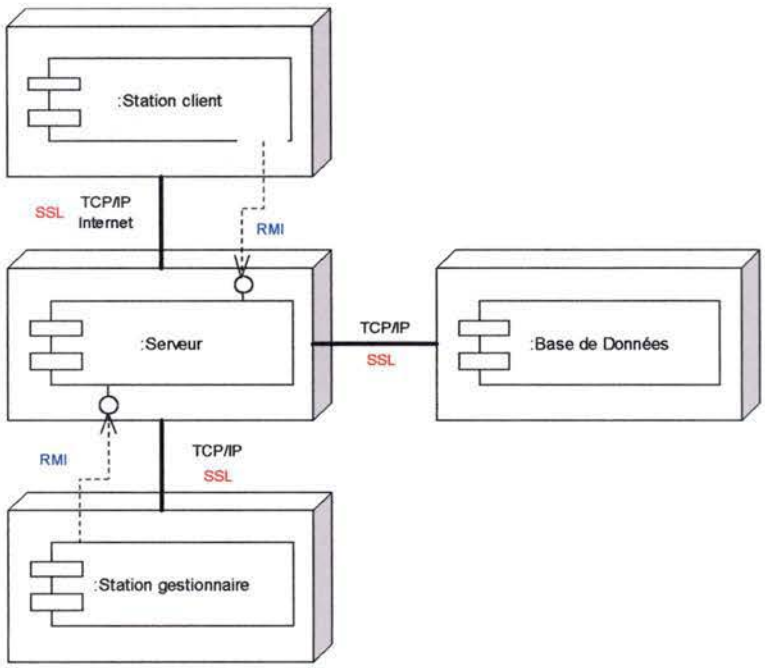


FIG. 88 – PNB : SCHÉMA CONCEPTUEL

Sur base de l'architecture logique du système et du diagramme de déploiement des composants, la question à se poser est de savoir quels sont les risques de sécurité encourus en terme d'accès illicite ou d'attaques contre le système.

Pour cela, nous distinguons deux points de vue fondamentaux dans l'analyse de sécurité du système : le point de vue du réseau et le point de vue des objets.

7.8.1. LE POINT DE VUE DU RÉSEAU.

Dans notre application, nous identifierons deux types de connexions physiques principales, suivant qu'elles concernent un support interne au réseau de l'entreprise (TCP/IP « intranet ») ou au contraire un support public (TCP/IP internet) :

- CONNEXION WORKSTATION CLIENT → SERVEUR D'APPLICATION

La particularité de cette connexion est qu'elle provient d'un réseau public externe à l'entreprise. Le premier élément de sécurité à introduire consiste à s'assurer que le client qui se connecte au serveur est bien celui qu'il prétend être. Il est également essentiel de s'assurer qu'aucune possibilité d'écoute indiscrète ne permet d'intercepter les communications entre le client et le serveur. Il apparaît donc incontournable d'introduire un mécanisme permettant d'*authentifier le client*, et de *garantir la confidentialité* des communications.

- CONNEXION WORKSTATION GESTIONNAIRE → SERVEUR D'APPLICATION
- CONNEXION MIDDLEWARE → SERVEUR DE BASE DE DONNÉES

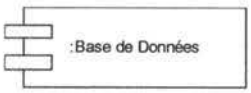
Ici, les connexions établies sont limitées au réseau interne de l'entreprise. En pratique, il est déjà possible d'agir au niveau de la construction physique du réseau (par exemple en segmentant le réseau afin de s'assurer qu'aucune écoute n'est possible entre segments). Si nous supposons que le réseau n'est pas fiable du point de vue de l'écoute, il est également nécessaire de *garantir la confidentialité des données en transit sur le réseau*.

Compte tenu des technologies Java que nous avons étudiées, il semble que la solution évidente à ces différents besoins consiste en l'utilisation du protocole SSL. En effet ce dernier permet d'assurer un haut niveau de confidentialité d'un point de vue du trafic réseau, indépendamment des informations susceptibles de transiter sur ce réseau, et indépendamment du type de réseau (privé ou public) concerné. En outre, SSL couvre également le problème d'authentification, tant du client que du serveur, sur base de certificats numériques. Enfin, d'un point de vue Java, nous avons vu que SSL fait partie intégrante des techniques de sécurité, et qu'il peut être utilisé avec des technologies d'accès à distance de type RMI.

7.8.2. LE POINT DE VUE DES OBJETS.

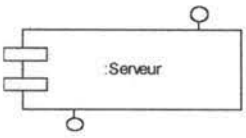
Après s'être penché sur les aspects de la sécurité d'un point de vue réseau, il convient d'étudier les différents composants du système (point de vue des objets) :

- La base de données ;
- Le middleware (serveur d'application) ;
- Le client / le gestionnaire.

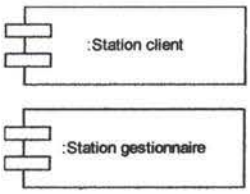


La **base de données** est presque toujours un composant clé en matière de sécurité ; c'est évidemment le cas lorsqu'il s'agit d'une application bancaire. Une base de données doit garantir que seuls les accès autorisés sont possibles. Elle doit aussi pouvoir garantir le secret de certaines données sensibles, qui ne peuvent être lues clairement même par les personnes autorisées à accéder à la base de données (c'est par exemple le cas des codes d'accès utilisateurs).

Il est clair qu'une première approche réseau impose de restreindre physiquement (c'est-à-dire par la configuration du matériel) l'accès au serveur à certaines machines du réseau bien identifiées et en nombre le plus limité possible. D'autres mesures sont également à envisager, telles que des mesures propres au système de gestion de base de données lui-même (contrôles d'accès). Mais au-delà de ces premières mesures qui ne concernent pas l'aspect Java de la sécurité, la base de données exige l'*authentification* des clients (au sens client-serveur) et la *confidentialité* de certaines données.



Le **serveur d'application** occupe un rôle central dans l'architecture du système ; c'est par lui que sont accessibles les différents services offerts, et c'est également par lui que transitent les demandes d'accès au serveur de base de données. Dans notre architecture, le serveur d'applications est directement exposé aux connexions des clients, internes ou externes. Il est donc important de s'assurer que tous les accès au middleware sont autorisés. Cela implique aussi de pouvoir vérifier qu'un utilisateur authentifié a le droit d'accéder à un service particulier. Par conséquent, en plus d'un mécanisme d'*authentification* des clients, il faut mettre en place une politique de *gestion des accès*.



Le **client** et le **gestionnaire** doivent disposer d'une partie d'application pour pouvoir utiliser le système. La partie client doit être la plus réduite possible, afin de limiter au maximum tout risque de manipulation locale de l'application, mais aussi pour faciliter le déploiement de l'application et sa maintenance. En outre, le client doit disposer des outils nécessaires pour son identification auprès du serveur (nom d'utilisateur et mot de passe). Il doit pouvoir authentifier le serveur bancaire auquel il se connecte. De plus, les informations transmises durant la connexion peuvent avoir un caractère confidentiel (données privées de connexion, données bancaires, etc). Le client doit également s'assurer que le serveur ne dispose pas aveuglément des ressources de la machine locale, et que le code exécuté sur sa machine est bien celui de la banque. Le client doit donc disposer d'outils d'*authentification*, d'outils de gestion de la *confidentialité* et d'outils de *gestion des accès* aux ressources de sa machine (par exemple des connexions réseaux, ou encore de l'accès à l'imprimante).

7.8.3. IMPLEMENTATION DU CLIENT ET DU GESTIONNAIRE

La banque a préalablement fournit un module de lancement au client (interne ou externe), qui comprend les classes de connexion RMI et le chargeur de classe personnalisé. Il doit aussi disposer d'un certificat client (inclus dans un **keystore**), un nom d'utilisateur et un mot de passe initial.

Le client doit bien entendu disposer de sa propre clé privée, correspondant à la clé publique enregistrée dans le certificat qui sera envoyé à la banque.

Le client doit également disposer de la clé publique de la banque. Celle-ci doit être lue dans un certificat fourni par la banque et signé par une autorité de certification. Ce certificat peut être chargé au départ du serveur bancaire lors de l'établissement de la connexion ; cette technique nécessite la présence sur le poste client d'un certificat de confiance (« *root certificat* », voir *3.4.3. Communication SSL*, « *TrustManagerFactory* ») permettant la validation du certificat bancaire. C'est ainsi que fonctionnent les navigateurs internet de façon quasiment transparente. On peut également supposer l'enregistrement préalable par la banque du certificat contenant sa clé publique dans un fichier **keystore** sur la machine cliente ; pour la simplification de l'exemple, et comme ce dernier n'est pas destiné à être exécuté à travers un navigateur, c'est cette alternative que nous allons mettre en œuvre.

A l'aide du module de lancement, le client établit une connexion RMI avec le serveur bancaire, et met en place une connexion de type **SSL** en s'authentifiant avec le certificat client contenu dans le keystore. A partir de ce point, tout appel de service se fait via un appel RMI sécurisé.

Une fois la connexion SSL établie, le client effectue son login en fournissant son nom d'utilisateur et son mot de passe. Notons que ces informations circulent déjà de façon chiffrée sur le réseau.

Lorsque l'utilisateur a été identifié par le système, un **chargeur de classe personnalisé** peut être mis en œuvre sur la machine cliente. Ce chargeur permet l'utilisation du module client, à savoir la gestion du menu client qui utilisera la connexion RMI. Soulignons que ce module client est chargé à travers la connexion sécurisée SSL. Le chargeur de classe peut en outre effectuer un contrôle de la **signature du code** téléchargé, afin de s'assurer que le code provient bien du serveur de la banque.

Le client charge une classe Menu, et reçoit un objet Menu qui correspond à son type d'utilisateur (client ou gestionnaire). Cet objet Menu constitue un **objet gardé**, associé avec des **permissions** accordées à l'origine du code (serveur de la banque). Ces permissions concernent par exemple la gestion d'une connexion réseau de type RMI sur un port particulier, ou encore l'accès à l'imprimante (pour l'impression de l'historique du compte).

CONCLUSION

SÉCURITÉ : MYTHE OU RÉALITÉ ?

Indépendamment de l'aspect technique, le point faible de toute sécurité réside en réalité dans un aspect humain et social. En effet, si une application quelconque peut disposer de clés de chiffrement très difficiles à casser, elle n'est pas pour autant à l'abri de toute investigation. Comme nous l'avons vu, l'ensemble de l'infrastructure repose essentiellement sur la notion de tiers de confiance, sorte de « gardien des clés ». C'est là le maillon faible du dispositif car, techniquement, ces clés centralisées peuvent parfaitement permettre de s'introduire dans tout système de sécurité sans effort, ne serait-ce que sous couvert d'une autorité gouvernementale. Les limites de la sécurité sont donc aussi affaires de dispositions légales...

Il est donc illusoire de considérer la sécurité en général - et la cryptographie en particulier – comme des moyens absolus de confidentialité, d'authentification ni même d'intégrité, mais plutôt comme des moyens nécessaires et la plupart du temps suffisant pour garantir ces objectifs dans des applications industrielles et commerciales. En fournissant les outils d'une communication plus sûre, l'informatique moderne offre à Big Brother de nouvelles armes sur un plateau d'argent.

Un autre aspect fondamental de la sécurité et de la cryptographie est son origine mathématique, qui lui confère une aura de sécurité totale puisque mathématiquement démontrée. Mais que peut une démonstration mathématique face à une mauvaise mise en œuvre, à une programmation défaillante, à un tiers de confiance non fiable ou à une autorité gouvernementale curieuse ? Comment garantir l'absolue maîtrise de processus de plus en plus complexes, faisant intervenir de plus en plus de composants de plus en plus interconnectés ?

Et même en admettant que le système conçu soit une merveille de processus sécurisé, comment être sûr de l'environnement dans lequel ce système va évoluer ? Que dire de la maîtrise du système d'exploitation sur lequel il va s'exécuter, de toute la plateforme matérielle, de l'infrastructure réseau locale et publique, des personnes mêmes qui vont être confrontées à ce système ?

L'infaillibilité mathématique elle-même n'est pas absolue. Comme nous l'avons vu dans « Problèmes et limites de la cryptographie », l'invulnérabilité des algorithmes est aussi à considérer dans une perspective temporelle. Les algorithmes de chiffrement invulnérables d'hier ne le sont plus forcément aujourd'hui, ne serait-ce que parce que la capacité des machines à utiliser la force brute pour casser un chiffrement a augmenté de manière considérable.

Bien sûr, certains algorithmes sont prétendus indépendants, en pratique, de la puissance des machines, tant la difficulté d'utiliser la force brute est énorme. C'est par exemple le cas de l'algorithme RSA, basé sur la difficulté mathématique de factoriser des grands nombres. Mais qui peut nous garantir qu'il n'y aura pas demain un article de la presse spécialisée nous présentant le futur prix Nobel de mathématique qui aura découvert une technique miracle pour une factorisation efficace ?

En 1996, Ariane V explosait en vol peu après le décollage. Dans un processus aussi complexe et surveillé que la construction d'un programme spatial, il a suffit du débordement d'un registre de 16

bits hérité d'Ariane IV pour volatiliser un petit milliard d'euros en plein ciel de Guyane (source : le numéro de décembre 1997 du périodique du CNRS français consacré à la sécurité informatique). Un grain de sable dans une gigantesque mécanique sous contrôle draconien. Il en va de même pour les systèmes d'informations prétendument sécurisés : qui peut vraiment dire quelle est la faille du système avant que celle-ci n'ait été mise au jour parfois de façon spectaculaire – et coûteuse !

SÉCURITÉ DANS JAVA : NOTRE DÉMARCHE

Nous présentons ici la démarche générale que nous avons poursuivie pour la construction de ce mémoire. Le point de départ de notre approche a été l'étude des aspects de sécurité dans Java, à travers ses architectures et ses extensions.

Cette étude nous a poussés naturellement à nous intéresser à la sécurité des systèmes d'informations en général, aux services de sécurité reconnus, aux principaux moteurs cryptographiques disponibles, et aux algorithmes standards.

Ces investigations nous ont conduits à détailler plus en profondeur certains mécanismes originaux proposés par Java en matière de sécurité. Chacun de ces mécanismes a été décrit dans les différents « zooms » de le chapitre *3. Sécurité dans Java*.

Devant l'aspect distribué des applications, nous nous sommes également intéressés à l'ensemble des technologies distribuées en les comparant à la technologie présente dans Java.

Ceci nous a notamment permis de mieux éclairer l'évolution constante de RMI vers CORBA, qui se confirme encore avec les récentes spécifications de Sun considérant comme obsolète pour RMI le protocole propriétaire JRMP au profit du protocole ouvert IIOP issu de CORBA.

A travers les différents documents que nous avons étudiés, nous avons constaté que les techniques de sécurité seules ne peuvent être mises en œuvre sans être prises en compte lors d'une démarche de conception des applications. C'est donc vers la modélisation des applications orientées objets que nous avons orienté la seconde partie de ce mémoire.

Devant le peu de préoccupation en matière de sécurité dans la norme UML, nous avons cherché d'éventuels travaux quant à l'introduction des aspects de sécurité dans le standard de l'OMG. C'est ainsi que nous avons découvert les récents travaux du professeur Jürjens d'une part, et du professeur Brose d'autre part, chacun exprimant des idées originales permettant d'intégrer la sécurité dans UML en réutilisant la notation usuelle.

Forts de notre acquis technique et méthodologique, et devant l'absence de démarche générale de développement sécurisé, nous avons essayé de dégager un guide de conception basé sur UML et fédérant l'ensemble des idées présentées. Un exemple d'application illustre les idées exposées.

Ainsi, partant d'un niveau purement technique, notre démarche nous a conduits à un niveau méthodologique permettant de replacer les techniques de sécurité en général – et celles de Java en particulier – dans une démarche globale de développement.

FAUSSES PISTES...

La construction dynamique de notre démarche, caractéristique d'un mémoire d'investigation, nous a conduits à explorer dès le départ un grand nombre de pistes sans pouvoir juger au préalable de leur intérêt réel. Ceci a eu deux conséquences essentielles : d'une part le foisonnement d'informations, parfois incohérentes ou contradictoires, dont il a fallu extraire la substantifique moelle, et d'autre part le développement parfois inutile de fausses pistes.

La principale fausse piste que nous ayons suivie concernait la représentation graphique des notations de sécurité dans UML. Devant l'absence apparente d'outil prenant en compte ces considérations, nous avons envisagé de développer une interface graphique basée partiellement sur les travaux de Jan Jürjens, que nous aurions pu compléter avec une notation personnelle. Un noyau d'interface a même été construit dans ce sens.

L'analyse approfondie des notations proposées par Jürjens et Brose a montré qu'elles se satisfaisaient d'extensions standards d'UML en les mettant à profit pour exprimer des aspects de sécurité. L'intérêt manifeste d'une notation standard par rapport à une notation personnelle a conduit à l'abandon de cette piste.

... ET PISTES À SUIVRE

Au-delà des pistes abandonnées, et compte tenu de l'étendue du sujet, il est évident que nous n'avons pas pu explorer toutes les pistes utiles ou intéressantes. Parmi celles-ci, nous épinglerons trois pistes qui nous semblent particulièrement pertinentes.

Une **première piste à suivre** est issue d'une affirmation répétée tout au long de ce mémoire. En effet, nous avons insisté à plusieurs reprises sur l'importance de considérer la sécurité dès les premières phases de développement d'un nouveau projet Java. On peut toutefois se poser la question de savoir comment sécuriser un projet existant non sécurisé. Il serait intéressant de s'intéresser à une phase de rétro-ingénierie sur le projet, pour se "raccrocher" à notre guide sécurisé. Si cette phase n'est pas possible (ne serait-ce que pour une question de coût), quelles sont les possibilités ? Il doit être possible de reconstituer l'équivalent de la dernière étape du guide, concernant l'architecture de l'application.

Quoi qu'il en soit, la solution vient notamment du fait que Java cache une partie de ses mécanismes de sécurité; en effet, au lieu de parler d'application "non sécurisée", il faudrait parler en réalité d'application "pré-sécurisée par défaut", c'est à dire utilisant les mécanismes de sécurité opérant lorsque aucune autre mécanisme n'est explicitement défini (l'exemple le plus éloquent est à ce propos la *sandbox* de la JVM).

Il est aussi important de se rendre compte que Java encapsule certains services de sécurité de telle sorte qu'il est possible de les utiliser quasiment indépendamment du reste de l'application. Par exemple, nous avons vu qu'une application distribuée construite sur SSL (voir Zoom 3) pouvait aisément utiliser une communication sécurisée grâce à SSL : il suffit pour cela de remplacer les sockets de RMI par des sockets sécurisés SSL. Le reste de la communication RMI reste transparente.

De la même manière, grâce au chargeur de classe personnalisable, il est possible d'ajouter des contrôles de chargement du code (origine, authenticité, etc.) en insérant un classloader dans la chaîne des classloader existants, sans perturber l'exécution générale du code (puisque les classes qui ne sont pas concernées par le nouveau classloader continueront à être chargées comme avant).

Par contre, ajouter un Contrôleur d'Accès efficace n'est pas évident sur une application existante qui ne prend pas en compte ce genre de besoins (ne serait-ce que parce que le code ne prendra pas en compte les exceptions de sécurité appropriées). Ce genre de contrôle fin demande en effet une analyse très poussée, qui s'étale tout au long du processus de notre Guide. Sans une phase de rétro-ingénierie complète, il semble difficile de garantir la même granularité au niveau des permissions d'accès. De plus, il faut juger de l'importance de l'impact sur le code de l'ajout des contrôles de permission.

Une **deuxième piste à suivre**, d'un grand intérêt, est liée à l'encapsulation des services de sécurité initiée par Java. Il serait très intéressant de pouvoir encapsuler réellement certains services de sécurité complets, qui deviendraient réellement indépendants du reste du code de l'application (une sorte de "boîte noire" qui offrirait des services de sécurité sans avoir à en connaître les détails techniques). Un exemple évident est encore une fois RMI-SSL : il ne serait sans doute pas très compliqué d'encapsuler une fois pour toute la création des sockets sécurisés pour RMI.

Toute la difficulté consiste plutôt dans une paramétrisation efficace et dans une réelle réflexion sur la généricité du code. Un autre point clé réside dans le bon équilibre entre réutilisabilité et sécurité, puisque nous avons vus dans le Guide (voir *6.3. Règles de Programmation Java*) que la réutilisabilité est parfois l'ennemie de la sécurité.

Une **troisième piste à suivre** serait de pousser plus avant l'investigation dans CORBA. En effet, nous avons vu que le rapprochement de RMI vers CORBA via RMI-IIOP est une réalité de plus en plus tangible. Il serait par conséquent intéressant de voir si les principes de sécurité décrits pour les applications distribuées Java s'appliquent de la même manière pour les applications CORBA, voire de proposer un guide pour migrer une application RMI vers une application CORBA sécurisée.

Ainsi, par exemple, les travaux de [BROSE 01] proposent des outils de traduction semi-automatique des contrôles d'accès en langage IDL au départ du diagramme de use case.

Comme on peut le voir à travers ces simples exemples – loin d'être exhaustifs – de pistes à suivre, la matière à investigation ne manque pas.

ÉVALUATION DES OBJECTIFS-PHARES

Au terme de ce mémoire, il nous reste à reconsidérer les objectifs-phares que nous avons énoncés lors de l'introduction.

Le premier de ces objectifs concernait l'étude des techniques de sécurité liées à l'environnement Java. Comme prévu initialement, la rencontre de cet objectif a occupé une grande partie du temps consacré au mémoire. Nous avons assimilé une grande quantité d'informations, comparé les différentes sources, ajouté de la précision à certains endroits où elle faisait défaut. Étant donné l'ampleur du sujet, l'étude des techniques Java n'a évidemment pas été exhaustive. L'objectif premier de ce mémoire nous semble pourtant pleinement atteint.

Le second objectif-phare concernait une description pédagogique des techniques étudiées. Il nous semble que ce mémoire constitue au moins une bonne vue d'ensemble du sujet, cohérente et mise à jour, aussi bien des fondements de l'architecture Java et de ses possibilités originales, que des concepts de sécurité en général dans un contexte distribué. L'objectif pratique de toute démarche pédagogique est renforcé par le guide de sécurisation. Il nous paraît susceptible d'éveiller l'intérêt du lecteur par les nombreuses pistes qu'il met en évidence. Un bémol doit toutefois être apporté à l'aspect pédagogique avec l'application exemple qui n'a pu être finalisée, faute de temps. Elle aurait pourtant constitué un complément appréciable à notre travail.

Le troisième objectif-phare considérait les aspects de la sécurité dans un cadre distribué. C'est une des motivations qui nous a poussé vers les architectures distribuées autres que Java. Nous avons pu ainsi effectuer un survol de ces technologies, suffisant pour pouvoir en extraire les grandes caractéristiques, et tenté d'en effectuer un comparatif. Toutefois, il est bien évident que cette étude n'était pas destinée à choisir une technologie pour la suite du mémoire, mais seulement à replacer Java dans un cadre plus général. A ce titre, rappelons une des pistes à suivre déjà citée : il serait très intéressant en effet de développer la partie qui concerne l'architecture CORBA.

La réalisation du quatrième objectif-phare ne fait aucun doute, puisqu'il concernait la définition de lignes de conduite dans la sécurisation d'un projet Java. Cependant, comme nous l'avons fait remarqué dans les « pistes à suivre » ci-dessus, il nous semblait intéressant d'étudier ces pistes dans le cadre de la sécurisation d'une application Java existante. Malheureusement, le temps nous a manqué pour compléter cette étude.

Dans l'ensemble, il nous semble donc que l'essentiel des objectifs a été atteint, même si, par manque de temps, certaines parties n'ont pu être développées plus avant.

Cette conclusion ne serait pas complète sans évoquer l'enrichissement personnel consécutif à la réalisation de ce mémoire.

D'un côté, il faut évoquer les difficultés d'organisation inhérentes au travail en équipe : synchronisation des horaires de travail, déplacements, locaux et équipements, etc.

Toutefois, devant la documentation pléthorique et la complexité de la matière à étudier, l'avantage du travail à deux apparaît dans la confrontation permanente des idées et des compréhensions. La recherche permanente de consensus qui s'ensuit conduit à une dynamique de travail qui garanti une meilleure cohérence globale.

Nous tenons à insister ici sur l'intérêt constant qui a été la principale motivation à l'élaboration de ce travail, et sans laquelle il ne nous aurait sans doute pas été possible de le mener à bien.

REFERENCES

[ANDOH 99] **ANDOH** Akira, **NASH** Simon - *RMI over IIOP*, JavaWorld.com, USA, 12.1999. - http://www.javaworld.com/

[BLAKLEY 99] **BLAKLEY** Bob - *Corba Security, An Introduction to Safe Computing with Objects*, ADDISON-WESLEY, « Booch,Jacobson,Rumbaugh series », US, 11.1999, ISBN 0-201-32565-9.

[BONJOUR 00] **BONJOUR** M., **FALQUET** G., **GUYOT** J., **LEGRAND** A. - *Java : de l'Esprit à la Méthode*, VUIBERT, Paris, FR, 04.2000 (2ème éd), ISBN 2-7117-8647-1.

[BROSE 01] **BROSE** G., **KOCH** M., **LÖHR** K. - *Integrating Security Policy into the Software Development Process*, Institut für Informatik Freie Universität Berlin, GE, 11.2001 - http://www.inf.fu-berlin.de/~brose

[CERT 97] **(Cert)** - *Security of the Internet*, CERT Coordination Center Reports, 01.1997. - http://www.cert.org/

[COUTURE 00] **COUTURE** Mario - *Description de Java RMI et comparaison avec CORBA et DCOM*, Université Laval, Computer Vision and Systems Laboratory, Québec, Canada, 08.2000. - http://www.gel.ulaval.ca/~vision/

[CRYPTOSEC 00] **(Cryptosec)** - *SSL-TLS : Sécurisation de flux applicatifs* - FR, 2000. - http://cryptosec.lautre.net/

[CURTIS 99] **CURTIS** Dave - *RMI, IIOP et EJB*, Inprise Corporation, CA, USA, 04.1999. - http://www.borland.com/

[DONSEZ 00] **DONSEZ** Didier, **BOURZOUFI** Hafid, - *RMI - Remote Method Invocation*, Université de Valenciennes, France, 2000. - http://www.univ-valenciennes.fr/

[FOWLER 01] **FOWLER** Martin - *UML*, CAMPUS PRESS, Paris, FR, 11.2001 (Original Title : *UML Distilled, 2cd Edition*, Addison-Wesley Longman Inc, USA, 2000), ISBN 2-7440-1336-6.

[FUNDP 00] **(Fundp)** - *Laboratoire de développement de logiciels IHDC2205* - Cours de N.HABRA et V.ENGLEBERT - Facultés Universitaires Notre-Dame de la Paix, Namur, BE, 2000-2001. - http://fundp.info.ac.be/

[GARMS 01] **GARMS** Jess, **SOMERFIELD** Daniel - *Professional Java Security*, WROX Press Ltd., Birmingham, UK, 05.2001, ISBN 1-8610-0425-7.

[GEIB 98] **GEIB** Jean-Marc, **GRANSART** Christophe, **MERLE** Philippe - *CORBA : des concepts à la pratique* - Université des Sciences et Tecnologies de Lille , FR, 1998. - http://corbaweb.lifl.fr/

[GONG 98A] **GONG** Li, **SCHEMERS** R. - *Implementing Protection Domain in the JDK 1.2* Technical report, JavaSoft, Sun Microsystems Inc., US, 1998. - http://java.sun.com/security/

[GONG 98B] **GONG** Li, **SCHEMERS** Roland - *Signing, Sealing and Guarding Java Objects*, Technical report, JavaSoft and Sun Microsystems Inc., US, 1998. - http://java.sun.com/security/

[HORSTMANN 00] **HORSTMANN** Cay S., **CORNELL** Gary - *Au Coeur de Java 2 : Fonctions avancées*, CAMPUS PRESS, Paris, FR, 06.2000, ISBN 2-7440-0881-8.

[JAWORSKI 00] **JAWORSKI** Jamie, **PERRONE** Paul - *Java Security* , CAMPUS PRESS, Paris, FR, 09.2001 (Original Title : *Java Security Handbook*, SAMS Publishing, Indianapolis, USA, 10.2000), ISBN 2-7440-1230-0.

[JURJENS 01] **JÜRJENS** Jan - *Secure Java development with UML*, in *Advances in Network and Distributed Systems Security*, 1st Annual Working Conference on Network Security, Leuven, BE, 11.2001, Kluwer Academic Publishers, 2002.

[JURJENS 01B] **JÜRJENS** Jan - *Developing Secure Systems with UMLsec – From Business Processes to Implementation*, VIS 2001, Kiel ,AL, 09.2001 - http://www4.in.tum.de/~juerjens/

[JURJENS 01C] **JÜRJENS** Jan - *Formal Development and Verification of Security-Critical Systems with UML*, Workshop on Automated Verification of Critical Systems, Oxford, UK, 04.2001 - http://www4.in.tum.de/~juerjens/

[JURJENS 01D] **JÜRJENS** Jan - *Encapsulating Rules of Prudent Security Engineering*, International Workshop on Security Protocols, Cambridge, UK, 04.2001 - http://www4.in.tum.de/~juerjens/

[JURJENS 01E] **JÜRJENS** Jan - *Transformations for Introducing Patterns – A Secure Systems Case Study*, Workshop on Transformations in UML (ETAPS 2001 Satellite Event), Genoa, 04.2001 - http://www4.in.tum.de/~juerjens/

[JURJENS 01F] **JÜRJENS** Jan - *Towards Secure Systems Development with UMLsec*, Fundamental Approaches to Software Engineering (FASE/ETAPS 2001, Genoa 04.2001 - http://www4.in.tum.de/~juerjens/

[JURJENS 02] **JÜRJENS** Jan - *Using UMLsec and Goal Trees for Secure Systems Development*, Symposium of Applied Computing (SAC 2002), Madrid, 03.2002 - http://www4.in.tum.de/~juerjens/

[LOPEZ 98] **LOPEZ** N., **MIGUELS** J., **PICHON** E. - *Intégrez UML dans vos projets*, EYROLLES, Paris, FR, 1998, ISBN 2-212-08952-X.

[MAIWALD 01] **MAIWALD** Eric - *Sécurité des réseaux*, CAMPUS PRESS, Paris, FR, 12.2001, ISBN 2-7440-1240-8.

[MCGRAW 98] **MCGRAW** G., **FELTEN** E. - *12 rules for developing more secure Java code* - US, 12.1998. - http://www.javaworld.com/

[NAESSENS 01] **NAESSENS** Vincent, **VANHAUTE** Bart, **DE DECKER** Bart - *Securing RMI Communication*, in *Advances in Network and Distributed Systems Security*, 1st Annual Working Conference on Network Security, Leuven, BE, 11.2001, Kluwer Academic Publishers, 2002.

[NETSCAPE 98] **(Netscape)** - *Introduction to SSL*, Netscape - US, 1998. - http://developer.netscape.com/docs/manuals/security/

[NIEMEYER 00] **NIEMEYER** Patrick, **KNUDSSEN** Jonathan - *Introduction à Java*, Editions O'REILLY, Paris, FR, 11.2000, (Original Title : *Learning Java*, O'REILLY & Associates Inc., UK, 2000), ISBN 2-84177-127-X.

[OAKS 01] **OAKS** Scott - *Java Security*, O'REILLY & Associates Inc., UK, 05.2001, ISBN 0-5960-0157-6.

[SOLERI] **(Soleri)** - *Java et les objets distribués*, Paris, FR - http://www.soleri.com/

[SRINIVAS 00] **SRINIVAS** Raghavan - *Java Security evolution and concepts, Part 2* - US, 07.2000. - http://www.javaworld.com/

[SUN 02] **(Sun)** - *On Line Documentation*, Java 2 Platform St. Ed v.1.4.0 Sun Microsystems Inc, US, 2002 - http://java.sun.com/

[TANENBAUM 97] **TANENBAUM** Andrew - *Réseaux*, 3ème Ed. DUNOD / Prentice Hall, Paris, FR, 07.1997, ISBN 2-1000-4315-3.

[TRAVIS 01] **TRAVIS** Greg - *Using JSSE for secure socket communication* - US, 2001. - http://www.ibm.com/developerWorks/

[UNIGE 00] **(Unige)** - *Introduction à la cryptographie* Institut universitaire de hautes études internationales, Genève, CH, 07.2000 - http://heiwww.unige.ch/

[VIEGA 01] **VIEGA** John, **MCGRAW** Gary - *Building Secure Software*, Addison Wesley Professional, UK, 09.2001, ISBN 0-201-72152-X.

[WEBATRIUM 00] **(WebAtrium)** - *Architectures distribuées et serveurs d'application*, CALIDIS Group, 2000. - http://www.calidis.com/

[ZIFF 02] **DECKMYN** D., **VAN PARIDON** P., **HARTMAN** R. - *Dossier politiques de sécurité*, in *Smart Business*, Ziff Davis, US, 01.2002.