La protection dans les systèmes à objets répartis

Vincent Nicomette

Année 1996

Thèse

préparée au

Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS

en vue de l'obtention du

Doctorat de l'Institut National Polytechnique de Toulouse

Spécialité : Informatique

par

Vincent Nicomette

Ingénieur ENSEEIHT

La protection dans les systèmes à objets répartis

Soutenue le 17 Décembre 1996 devant le jury:

Président	**Jean-Claude**	**LAPRIE**
Rapporteurs	**Daniel P.**	**SIEWIOREK**
	Michel	**RAYNAL**
Examinateurs	**Bruno**	**d'AUSBOURG**
	Yves	**DESWARTE**
	Jean-Charles	**FABRE**
	Michel	**RIGUIDEL**
	Jean-Marc	**TOSQUES**

Cette thèse a été préparée dans le cadre du Laboratoire d'Ingénierie de la Sûreté de Fonctionnement, Laboratoire Coopératif: LAAS-CNRS, Aérospatiale, EDF, Matra Marconi Space, Technicatome, Thomson CSF

Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4

Rapport LAAS N° 96496

Avant-Propos

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS). Je remercie le Professeur Alain Costes, Directeur du Laboratoire, de m'avoir accueilli au sein du laboratoire.

J'exprime ma profonde reconnaissance à Jean-Claude Laprie, Directeur de Recherche CNRS, responsable du groupe "Tolérance aux fautes et Sûreté de Fonctionnement informatique" (TSF) et directeur du "Laboratoire de l'Ingénierie de la Sûreté de Fonctionnement" (LIS), pour m'avoir permis de mener mes travaux au sein du LIS.

Je tiens à remercier particulièrement Yves Deswarte, Directeur de Recherche INRIA, qui m'a encadré tout au long de cette thèse. Mes multiples intrusions dans son bureau ont toujours été conclues par des réponses à mes nouvelles questions, ses remarques pertinentes et ses critiques constructives m'ont toujours éclairé et m'ont permis de mener à bien cette thèse.

J'exprime ma vive reconnaissance et ma profonde amitié à Jean-Charles Fabre, Chargé de Recherche INRIA. Il m'a fait découvrir le monde de la recherche lors de mon stage de DEA et m'a fortement encouragé à entreprendre cette thèse. Sans lui, elle n'aurait jamais vu le jour. Il a de plus toujours été présent dans les moments difficiles, à la fois par ses conseils avisés sur le plan scientifique et par ses témoignages d'amitié.

J'exprime ma profonde gratitude à Jean-Claude Laprie pour l'honneur qu'il me fait en présidant mon Jury de Thèse, ainsi qu'à :

- Daniel P. Siewiorek, Professeur à l'Université de Carnegie Mellon ;
- Michel Raynal, Professeur à l'Université de Rennes ;
- Bruno d'Ausbourg, Ingénieur de Recherche du Centre d'Etudes et de Recherches de Toulouse (CERT) de l'Office National d'Etudes et de Recherches Aérospatiales (ONERA) ;
- Yves Deswarte, Directeur de Recherche INRIA au LAAS-CNRS ;
- Jean-Charles Fabre, Chargé de Recherche INRIA au LAAS-CNRS ;
- Michel Riguidel, ingénieur Thomson-CSF ;

- Jean-Marc Tosques, Adjoint au Directeur France (Direction des Systèmes Sols), Matra Marconi Space ;

pour l'honneur qu'ils me font en participant à mon jury. Je remercie particulièrement Daniel P. Siewiorek et Michel Raynal, qui ont accepté la charge d'être rapporteurs.

J'exprime ma gratitude à Daniel P. Siewiorek pour qui la langue dans laquelle est écrit ce manuscrit représente un surcroît de travail.

Je tiens à remercier tous ceux qui m'ont aidé ou soutenu tout au long de cette thèse avec une mention particulière pour l'ensemble des doctorants du groupe TSF et du LIS, Joelle Penavayre et Marie-José Fontagne pour leur efficacité et leur sourire, Michelle Powell et Anne Bergez pour leur disponibilité, Yves Crouzet qui m'a permis d'assouvir mes pulsions d'administrateur système, Eric Totel et Jean-Paul Blanquart pour leur relecture attentive de ce manuscrit et Tanguy Pérennou qui a l'agaçante habitude d'avoir toujours les bonnes idées aux bons moments.

Il m'est particulièrement agréable de remercier les membres du service II (Informatique et Instrumentation) auxquels j'ai très souvent fait appel. Le LAAS peut s'enorgueillir de posséder une équipe système très compétente et sympathique. Je tiens à citer plus particulièrement Marc Vaisset et Matthieu Herrb pour leurs réponses au nombre indécent de questions que j'ai posées durant ces trois années.

Je ne saurais oublier Emmanuel Chaput, doctorant du groupe OLC et Fabrice Francès, doctorant du groupe OLC et enseignant-chercheur a l'ENSICA, pour avoir toujours répondu patiemment à mes innombrables questions tordues en me donnant l'impression qu'elles étaient intéressantes.

Un grand merci à Evelyne, Françoise, Laurence, Muriel, Christophe, Jérome, Thierry, Tanguy et sa Tipo pour le partage des moments faciles et difficiles de ces trois années, ainsi qu'à toute la bande des copains toulousains.

Table des matières

Introduction générale

Depuis quelques années, il devient de plus en plus difficile de parler de systèmes informatiques sans parler de répartition, de réseaux et de communication. L'actuel engouement pour Internet et le développement des services que l'on peut y découvrir en est la preuve évidente. Le développement des systèmes répartis et les progrès techniques qui lui sont liés ont surtout concerné les fonctionnalités et la performance de ces systèmes: comment faire en sorte qu'il devienne de plus en plus facile et de plus en plus rapide de dialoguer, de s'informer, d'échanger de l'information à l'aide de systèmes informatiques? Quelles nouvelles fonctionnalités peut-on imaginer? Aussi, les travaux qui ont été menés ont consisté à faciliter l'accès aux systèmes informatiques en améliorant constamment leurs interfaces (nous sommes passés en très peu de temps de telnet aux navigateurs d'aujourd'hui tel Netscape) et en améliorant également les performances des communications (de quelques centaines d'octets par seconde il y a quelques années, les débits de transfert d'information sont passés à plusieurs méga-octets par seconde aujourd'hui, ces débits étant favorisés par le développement de technologies nouvelles comme ATM).

Cependant, les travaux relatifs à la sécurité et plus particulièrement à la protection des systèmes informatiques n'ont pas été à la mesure de ces avancées technologiques. Ainsi, en matière de protection, les concepts auxquels nous nous référons aujourd'hui ont peu évolué depuis une vingtaine d'années. Par exemple, un système tel que Multics, vieux de plus de vingt ans, est toujours cité comme une des références lorsque l'on aborde le problème de la protection. Ainsi, depuis plusieurs années, les travaux menés dans le cadre de la protection consistent souvent à "boucher les trous" de sécurité dont les systèmes actuels sont gorgés plutôt que de développer de nouveaux principes de protection. On est ainsi amené à développer des outils tels que Secure NFS par exemple, qui sont en fait construits de façon à respecter globalement l'interface des versions non sécurisées de ces outils, et à y insérer de la protection

plus ou moins artificiellement. La sécurité n'a en fait pas évolué parallèlement à l'évolution fonctionnelle des systèmes informatiques.

Il est donc essentiel de nous pencher sur le problème de la sécurité des systèmes répartis et en particulier sur le problème de la protection dans ces systèmes. N'est-il pas nécessaire de repenser la protection plutôt que d'essayer de constamment combler des failles de sécurité? De plus, le développement de nouvelles méthodes de conception de systèmes telles que la conception orientée objets ne nous amène-t-il pas également à penser différemment le problème de la protection?

Nous nous proposons dans ce mémoire, de présenter une nouvelle approche qui permette d'assurer la protection de systèmes répartis modélisés sous forme d'objets. Cette approche est intéressante pour plusieurs raisons:

- elle permet d'assurer la protection d'un système réparti sans posséder de points durs;
- elle respecte au mieux le principe du moindre privilège: chaque entité active du système exécute une opération avec les privilèges nécessaires à la réalisation de cette opération et uniquement ceux-ci;
- elle est basée sur la définition de nouveaux droits dont la gestion est particulièrement aisée.

Le premier chapitre de ce mémoire est consacré à la présentation des différentes techniques classiques de protection. Ce chapitre tente de mettre en avant les problèmes qui surviennent dès lors que l'on désire assurer la protection de systèmes répartis. Il introduit également la notion de systèmes composés d'objets répartis et pose le problème de la protection dans ce type de systèmes.

Le chapitre 2 est consacré à la présentation d'un schéma d'autorisation dans les systèmes composés d'objets répartis. Nous expliquons les concepts nouveaux que nous avons dégagés et la façon dont ils permettent au mieux de protéger les systèmes répartis dans leur globalité, en évitant le piège consistant à penser que protéger indépendamment chaque partie du système revient à le protéger globalement.

Le chapitre 3 est lui dédié à la présentation d'un modèle de sécurité multiniveau adapté au modèle objet. Nous avons voulu élaborer un modèle moins

restrictif que le modèle de confidentialité multiniveau de référence qu'est le modèle de Bell et LaPadula. Nous présentons donc dans ce chapitre ce modèle ainsi que ses principales limites et nous montrons comment nous avons créé un modèle moins restrictif tout en empêchant les flux d'information illégaux dans un système. Le chapitre 3 vient montrer de plus que le schéma d'autorisation présenté dans le chapitre 2 n'est pas lié à une politique de sécurité particulière et qu'il peut être également envisagé dans le cadre de politiques obligatoires multiniveaux.

Le chapitre 4 enfin détaille une expérimentation que nous avons effectuée sur une architecture composée de machines à base de micro-noyaux. Le prototype réalisé met en œuvre les différents principes présentés dans les chapitres 3 et 4 et en démontre la faisabilité et l'efficacité par un exemple concret de leur intégration dans un système réel.

Chapitre 1

Les techniques classiques de protection

Dans ce chapitre, nous présentons tout d'abord les définitions relatives à la notion de sûreté de fonctionnement. Nous nous intéressons plus particulièrement à un de ses attributs, la sécurité, dont nous définissons les principales propriétés.

Nous présentons ensuite les différents moyens existants pour assurer la sécurité d'un système et nous nous intéressons plus particulièrement à la notion de protection dont le but est double : empêcher des opérations non autorisées dans un système mais aussi limiter la propagation d'erreurs dans un système. Nous présentons la vision classique de la protection des systèmes centralisés puis nous mettons en avant les différents problèmes qui surgissent lorsqu'il s'agit d'assurer la protection d'un système réparti. Nous décrivons ensuite dans la dernière partie de ce chapitre, l'ensemble des nouvelles notions à considérer si l'on désire assurer la protection de systèmes à objets répartis. Nous expliquons les modifications que la programmation d'objets amène à considérer et nous finissons en présentant les travaux de l'Object Management Group (*OMG*) en matière de représentation de systèmes composés d'objets répartis. Nous nous attachons en particulier aux aspects liés à la sécurité.

1.1 La sécurité-confidentialité : attribut de la sûreté de fonctionnement

1.1.1 La sûreté de fonctionnement

1.1.1.1 Définitions de base

La **sûreté de fonctionnement** d'un système informatique est la propriété qui permet à ses utilisateurs de *placer une confiance justifiée dans le service qu'il leur délivre* [ABC+96]. Le **service** délivré par un système est son comportement *tel qu'il est perçu* par son ou ses utilisateurs ; un **utilisateur** est un autre système (humain ou physique) qui interagit avec le système considéré.

Selon les applications auxquelles le système est destiné, l'accent peut être mis sur différentes facettes de la sûreté de fonctionnement, ce qui signifie que la sûreté de fonctionnement peut être vue selon des propriétés différentes mais complémentaires, qui permettent de définir ses *attributs* :

- le *fait d'être prêt à l'utilisation* conduit à la **disponibilité** ;
- la *continuité de service* conduit à la **fiabilité** ;
- la *non-occurrence de conséquences catastrophiques pour l'environnement* conduit à la **sécurité-innocuité** ;
- la *non-occurrence de divulgations non-autorisées de l'information* conduit à la **confidentialité** ;
- la *non-occurrence d'altérations inappropriées de l'information* conduit à l'**intégrité** ;
- l'*aptitude aux réparations et aux évolutions* conduit à la **maintenabilité**.

L'association, à la confidentialité, de l'intégrité et de la disponibilité vis-à-vis des actions autorisées, conduit à la sécurité-confidentialité.

1.1.1.2 Notions de fautes, erreurs, défaillances

Une **défaillance** du système survient lorsque le *service délivré dévie de l'accomplissement* de la **fonction** du système, c'est-à-dire de ce à quoi le système est *destiné*. Une **erreur** est la partie de l'état du système qui est *susceptible d'entraîner une défaillance*: une erreur affectant le service est une indication qu'une défaillance survient ou est survenue. La *cause adjugée ou supposée* d'une erreur est une **faute.**

Un système ne défaille généralement pas toujours de la même façon, ce qui conduit à la notion de *mode de défaillance*, qui peut être caractérisée selon trois points de vue: domaine de défaillance, perception des défaillances par les utilisateurs du système, conséquence des défaillances sur l'environnement du système.

Le *domaine de défaillance* conduit à distinguer :

- les **défaillances en valeur** : la valeur du service délivré ne permet plus l'accomplissement de la fonction du système ;
- les **défaillances temporelles** : les conditions temporelles de délivrance du service ne permettent plus l'accomplissement de la fonction du système.

La *perception des défaillances* conduit à distinguer, lorsqu'un système a plusieurs utilisateurs :

- les **défaillances cohérentes** : tous les utilisateurs du système ont la même perception des défaillances ;
- les **défaillances incohérentes** : les utilisateurs du système peuvent avoir des perceptions différentes des défaillances ; les défaillances incohérentes sont souvent qualifiées de **défaillances byzantines.**

Les *conséquences des défaillances* conduisent à distinguer :

- les **défaillances bénignes**, dont les conséquences sont du même ordre de grandeur que le bénéfice procuré par le service délivré en l'absence de défaillance ;

- les **défaillances catastrophiques**, dont les conséquences sont incommensurablement différentes du bénéfice procuré par le service délivré en l'absence de défaillances.

Les fautes et leurs sources sont extrêmement diverses. Les cinq points de vues principaux selon lesquels on peut les classer sont leur cause phénoménologique (fautes physiques ou fautes dues à l'homme), leur nature (fautes accidentelles ou fautes intentionnelles), leur phase de création ou d'occurrence (fautes de développement ou fautes opérationnelles), leur situation par rapport aux frontières du système (fautes internes ou fautes externes) et leur persistance (fautes permanentes ou fautes temporaires).

Une faute est **active** lorsqu'elle produit une erreur. Une faute active est soit une faute interne qui était préalablement *dormante* et qui a été activée par le processus de traitement, soit une faute externe. Une faute interne peut cycler entre ses états dormant et actif.

Une erreur peut être latente ou détectée: une erreur est *latente* tant qu'elle n'a pas été reconnue en tant que telle; elle est *détectée* par un algorithme ou un mécanisme de détection ou parce qu'elle produit une défaillance. Une erreur latente peut disparaître (c'est-à-dire être corrigée) avant d'être détectée. Par propagation, une erreur crée de nouvelles erreurs.

Une défaillance survient lorsque, par propagation, une erreur affecte le service délivré par le système, donc lorsqu'elle traverse l'interface système-utilisateur. La conséquence de la défaillance d'un composant est une faute interne pour le système qui le contient et une faute externe pour les composants avec lesquels il interagit.

Ces mécanismes permettent de compléter la "chaîne fondamentale"suivante:

... $\Rightarrow$ défaillance $\Rightarrow$ faute $\Rightarrow$ erreur $\Rightarrow$ défaillance $\Rightarrow$ faute $\Rightarrow$ ....

Ces flèches ne doivent pas être interprétées au sens strict: par propagation, plusieurs erreurs peuvent être créées avant qu'une défaillance ne survienne; une défaillance étant un événement se produisant à l'interface entre deux systèmes ou composants, une erreur peut conduire à une faute sans que l'on observe de défaillance si l'observation de la défaillance n'a pas lieu d'être effectuée, ou si elle ne présente pas d'intérêt.

Par exemple, la conséquence de l'*erreur* d'un programmeur est une *faute* (dormante) dans le logiciel écrit (instruction ou donnée fautive); lorsque cette

faute est sensibilisée, elle devient active et produit une *erreur*; lorsque les données erronées affectent le service délivré, une *défaillance* survient.

De même, un court-circuit qui se produit dans un circuit intégré est une *défaillance* du circuit intégré; la conséquence est une *faute* (connexion collée à une valeur binaire, modification de la fonction du circuit, etc.), qui restera dormante tant qu'elle ne sera pas activée par des signaux particuliers d'entrée du circuit (ce qui produira une erreur).

1.1.1.3 Moyens de la sûreté de fonctionnement

Les moyens de la sûreté de fonctionnement sont les méthodes et techniques permettant de fournir au système l'aptitude à délivrer un service conforme à l'accomplissement de sa fonction, et de donner confiance dans cette aptitude. Cet ensemble de méthodes et techniques peut être classé en :

- **prévention des fautes** : comment empêcher l'occurrence ou l'introduction de fautes; il est intéressant de noter que la notion de prévention des fautes relève de l'ingénierie "générale"et déborde donc largement le cadre de la sûreté de fonctionnement;
- **tolérance aux fautes** : comment fournir un service à même de remplir la fonction du système en dépit des fautes;
- **élimination des fautes** : comment réduire la présence (nombre, sévérité) des fautes;
- **prévision des fautes** : comment estimer la présence, la création et les conséquences des fautes.

Ces moyens de la sûreté de fonctionnement sont en fait dépendants les uns des autres, ce qui motive le fait que seule leur utilisation combinée peut conduire à un système qui soit sûr de fonctionnement. Ainsi, l'association entre élimination des fautes et prévision des fautes peut-être considérée comme la **validation** de la sûreté de fonctionnement, c'est-à-dire comment *avoir confiance* dans l'aptitude du système à délivrer un service conforme à l'accomplissement de sa fonction. De même, l'élimination des fautes est souvent étroitement associée à la prévention des fautes, l'ensemble constituant l'**évitement des fautes**, c'est-à-dire comment *tendre vers* un système sans faute.

1.1.2 La sécurité

Dans la suite de ce mémoire, nous nous intéresserons uniquement à la sécurité-confidentialité que nous appellerons simplement sécurité lorsqu'il n'y aura pas d'ambigüité avec la sécurité-innocuité.

Assurer la sécurité d'un système informatique consiste à assurer la prévention d'accès et de manipulation illégitime de l'information ainsi que la garantie d'accès et de manipulation légitime de l'information. Un utilisateur d'un système sûr de fonctionnement (au sens de la sécurité) ne doit pas pouvoir *lire*, *écrire*, *créer* ou *détruire* les informations de manière illégitime. Modifier illégitimement des informations signifie transgresser un certain nombre de propriétés de sécurité qui définissent qui peut accéder aux informations du système, comment les utilisateurs peuvent y accéder et quelles sont les opérations que les utilisateurs peuvent réaliser sur ces informations. Ces propriétés de sécurité font partie de la **politique de sécurité** (cf. §1.2).

Assurer la sécurité d'un système, c'est assurer que les propriétés retenues dans la politique de sécurité sont toujours vérifiées.

1.1.2.1 La confidentialité

La **confidentialité** peut être définie comme la capacité du système informatique à empêcher la divulgation d'informations, c'est-à-dire à faire en sorte que les informations soient inaccessibles (ou incompréhensibles) pour les utilisateurs non désignés comme autorisés à y accéder [Des91]. Ceci correspond à empêcher un utilisateur de consulter directement une information qu'il n'est pas autorisé à connaître mais aussi à empêcher un utilisateur autorisé à lire une information de la divulguer à un utilisateur non autorisé à y accéder. Le terme information doit être pris au sens le plus large : il recouvre non seulement les données et les programmes, mais aussi les flux d'information et la connaissance de l'existence de données, de programmes ou de communications. Assurer la confidentialité d'un système est donc un travail bien plus colossal qu'il n'y paraît à première vue. Il faut analyser tous les chemins que peut prendre une information dans le système pour s'assurer qu'ils sont sécurisés. C'est un travail fastidieux et souvent trop coûteux par rapport aux besoins réels ; aussi dans bon nombre de cas, on ne s'occupera d'assurer la sécurité que d'un certain nombre des chemins que peut prendre l'information.

Les attaques contre la confidentialité consistent à essayer d'obtenir des infor-

mations malgré les moyens de protection et les règles de sécurité. Ces attaques peuvent être passives ou actives. Les attaques passives consistent à accéder aux informations qui sont générées, transmises, stockées ou affichées dans des composants vulnérables du système informatique (voies de communication, mémoires ou disques, etc.). Par exemple, lorsqu'un utilisateur U se connecte sur une machine B depuis une autre machine A, une écoute passive peut permettre de connaître l'identité et le mot de passe de l'utilisateur U tapés depuis la machine A et transmis en clair jusque la machine B. Cette manipulation permettra à un intrus de se connecter sur la machine B en prenant l'identité de l'utilisateur U. Une attaque active nécessite quant à elle, une action d'un ou plusieurs individus dans le système. L'utilisation des canaux cachés [Lam73] peut permettre ainsi à un individu autorisé à accéder à des informations de les transmettre à un utilisateur non autorisé à y accéder. Il peut pour cela utiliser des canaux de mémoire (par exemple, réutilisation de zones mémoires non réinitialisées) ou des canaux temporels (modulation de l'utilisation des ressources communes comme par exemple l'unité centrale ou les disques). Nous reviendrons plus en détail sur la notion de canaux cachés dans le paragraphe 1.3.

Les attaques contre la confidentialité peuvent être des fautes intentionnelles (ce que l'on conçoit parfaitement) mais aussi des fautes accidentelles. Un bon exemple de divulgation d'informations par faute accidentelle est celui du courrier électronique : une erreur d'alias et vous annoncez à tout le personnel de votre entreprise que vous venez de crever un pneu de la jaguar du directeur alors que vous vouliez juste l'annoncer à une personne de confiance !

1.1.2.2 L'intégrité

L'**intégrité** peut être définie comme la capacité du système informatique à empêcher la corruption des informations par les fautes accidentelles ou intentionnelles. Dans le cas des fautes intentionnelles, les attaques contre l'intégrité visent soit à introduire de fausses informations soit à modifier ou à détruire des informations (c'est-à-dire provoquer des erreurs) pour que le service (inapproprié) délivré par le système produise un bénéfice pour l'attaquant, au détriment des utilisateurs autorisés. C'est typiquement le cas des fraudes informatiques. Bien sûr, l'attaquant essaiera en général de faire en sorte que les erreurs qu'il introduit ne soient pas détectables et que la défaillance qui en résulte ne soit pas facilement identifiable.

L'interception est un exemple d'attaque contre l'intégrité. Elle consiste à accéder à des données transmises sur des voies de communication et à les modifier (en les détruisant, en les modifiant ou en y insérant des messages). Un exemple classique d'interception concerne les messages échangés entre deux machines utilisant le protocole IP (*Internet Protocol*) [IP81]. Chaque message IP est constitué d'une adresse source (la machine émettant le message) et d'une adresse destination (l'adresse de la machine à qui est destiné le message). Des interceptions de ces messages peuvent être réalisées de façon à changer l'adresse source dans le message. La machine cible croit alors recevoir un paquet d'une machine alors que c'est une autre qui l'a émis.

Un *virus* est également un bon exemple d'attaque contre l'intégrité d'un système. Un virus est un programme qui, lorsqu'il s'exécute, se propage et se reproduit pour s'adjoindre à un autre programme (du système ou d'application). Le virus sera exécuté (souvent en fonction de certaines conditions) lorsque le programme sur lequel il s'est adjoint sera lancé. La propagation du virus est une attaque contre l'intégrité des programmes, l'exécution du virus pouvant par ailleurs avoir d'autres effets qui peuvent être des attaques contre la confidentialité, la disponibilité et l'intégrité des données [Fer90].

Soulignons enfin que l'intégrité des informations peut être mise en défaut à la fois par des fautes intentionnelles mais aussi par des fautes accidentelles.

1.1.2.3 La disponibilité

Un attaquant peut avoir simplement pour but d'empêcher le système de remplir un service approprié. Cette attaque est alors une attaque contre la **disponibilité** du système qui est appelée *déni de service*. Ces attaques sont étroitement liées aux attaques contre l'intégrité étant donné qu'elles consistent souvent à détruire volontairement de l'information, que ce soit des données, des messages ou des processus de traitement afin de rendre le système incapable de fournir un certain service attendu.

Le déni de service est souvent un problème très délicat à résoudre, surtout dans le cadre des systèmes répartis. Prenons l'exemple de deux machines A et B, qui se situent sur un même réseau local qui comporte également d'autres machines. Supposons que la machine B doive recevoir régulièrement des messages de la machine A pour remplir un certain service. Supposons alors qu'une machine C du même réseau local soit corrompue et inonde constamment le réseau de messages divers. Les messages de A n'auront quasiment aucune chance

d'atteindre la machine B à temps pour que celle-ci remplisse correctement ses fonctions. Ce problème devient plus difficile à résoudre si les machines A et B se situent sur des réseaux locaux différents [Sat89] à cause du nombre élevé des points d'accès au support de communication.

Enfin, il est important de souligner que la disponibilité doit tenir compte à la fois des fautes intentionnelles mais aussi des fautes accidentelles : un câble débranché par une manœuvre maladroite peut amener un serveur de fichiers à ne plus répondre. Remarquons enfin que, dans les techniques de conception classique de systèmes sécurisés, la disponibilité est souvent négligée ou n'est considérée que vis-à-vis des fautes intentionnelles.

1.2 Politiques de sécurité

L'ensemble des propriétés de sécurité que l'on désire assurer dans un système ainsi que la façon dont on va les assurer sont définies dans la politique de sécurité du système.

"La politique de sécurité d'un système est l'ensemble des lois, règles et pratiques qui régissent la façon dont l'information sensible et les autres ressources sont gérées, protégées et distribuées à l'intérieur d'un système spécifique" [ITS91]. Elle doit identifier les objectifs de sécurité du système et les menaces auxquelles celui-ci devra faire face.

Cette notion de politique de sécurité peut être raffinée en trois branches distinctes : les **politiques de sécurité physique, administrative et logique.** La première s'occupe de tout ce qui touche à la situation physique du système à protéger. En particulier, y sont définies les mesures contre le vol par effraction, le feu, les catastrophes naturelles, etc. Il est à noter que dans le cas des systèmes répartis, la sécurité physique devient de moins en moins efficace, en raison du nombre important de points d'accès au système (stations et moyens de communication). Le développement d'Internet, avec de plus en plus de gens connectés depuis leur domicile grâce à un ordinateur personnel en est la preuve évidente. Les procédures administratives traitent de tout ce qui ressort de la sécurité d'un point de vue organisationnel au sein de l'entreprise ; la sélection du personnel responsable de la sécurité des systèmes informatiques en fait partie. La politique de sécurité logique s'intéresse au contenu du système informatique. Cette politique est en charge de réaliser tous les contrôles d'accès logiques, elle doit spécifier *qui* a accès à *quoi* et dans

quelles circonstances.

La politique logique peut se décomposer en plusieurs phases. Chaque individu qui utilise un système sécurisé doit s'identifier et doit pouvoir prouver qu'il est bien la personne qu'il prétend être. Ces deux phases sont définies dans la **politique d'identification** et dans la **politique d'authentification**. Lorsqu'un utilisateur est identifié et authentifié, la **politique d'autorisation** doit spécifier quelles sont les opérations que cet utilisateur particulier peut réaliser dans le système.

La politique d'autorisation est définie d'une part par un ensemble de *propriétés de sécurité* qui doivent être satisfaites par le système, et d'autre part, par un *schéma d'autorisation*, qui présente les règles permettant de modifier l'état de protection du système. Par exemple, une propriété de sécurité pourra être "une information classifiée ne doit pas être transmise à un utilisateur non habilité à la connaître", alors qu'une règle du schéma d'autorisation pourra être "le propriétaire d'une information peut accorder un droit d'accès pour cette information à n'importe quel utilisateur". Si la politique d'autorisation est cohérente, il n'est pas possible, partant d'un état initial sûr (c'est-à-dire pour lequel les propriétés de sécurité sont satisfaites), d'atteindre un état d'insécurité (c'est-à-dire pour lequel les propriétés de sécurité ne sont pas satisfaites) en appliquant les règles du schéma d'autorisation.

Les politiques d'autorisation, ou plus précisément leurs schémas d'autorisation, se classent en deux catégories: les **politiques discrétionnaires** et les **politiques obligatoires.**

Dans le cas d'une politique discrétionnaire, les droits d'accès à chaque information sont manipulés librement par le responsable de l'information (généralement le propriétaire), à sa *discrétion.* La gestion d'accès aux fichiers Unix constitue un exemple de mécanismes de contrôles d'accès basés sur une politique discrétionnaire. Dans ce cas, trois types d'accès possibles sont définis: *read*, *write* et *execute*. Le propriétaire d'un fichier peut librement attribuer ou non ces droits à lui-même, à un groupe d'utilisateurs, et aux autres utilisateurs. Mais imaginons qu'un utilisateur U1 ait confiance en un utilisateur U2 mais pas en un utilisateur U3. U1 donne donc les droits de lecture à U2 sur un de ses fichiers F mais pas à U3. U2 peut alors faire une copie de F (puisqu'il a le droit de le lire) et donner le droit de lecture sur cette copie à U3. Il est donc impossible, avec une politique discrétionnaire de contrôler ce type de fuite d'informations. De même, une politique discrétionnaire ne permet pas de résoudre le problème des chevaux de Troie. Un cheval de Troie [Gas88], est

un programme qui, sous couvert de réaliser une action connue, réalise à l'insu de la personne qui l'utilise, une autre action qui peut consister en une attaque contre la sécurité du système. Par exemple, sous Unix, si l'on réussit à placer un programme dont le nom est celui d'une commande standard (comme `ls`) dans un chemin tel qu'un utilisateur l'exécute (mauvaise utilisation de la variable d'environnement `PATH`), alors ce dernier, pensant exécuter la vraie commande, exécute en réalité un programme qui peut par exemple ajouter les droits en écriture sur son répertoire personnel.

Pour résoudre ce type de problème, les politiques dites "obligatoires" imposent, par leur schéma d'autorisation, des règles incontournables qui s'ajoutent aux règles discrétionnaires. Une politique obligatoire suppose que les utilisateurs et objets aient été étiquetés. Classiquement, les objets se voient assigner une classification, tandis que les utilisateurs possèdent une habilitation. Les règles qui régissent les autorisations d'accès sont basées sur une comparaison de l'habilitation de l'utilisateur et de la classification de l'objet. Ces règles incontournables assurent que le système vérifie des propriétés générales de confidentialité ou d'intégrité par exemple. Ces règles sont souvent utilisées conjointement aux règles d'une politique discrétionnaire car leur pouvoir d'expression est en général relativement faible. Dans ce cas, un utilisateur sera autorisé à manipuler une information dans le système si le droit de lecture est positionné sur l'information pour lui (contrôle discrétionnaire) et s'il est habilité à la manipuler (contrôle obligatoire).

La politique obligatoire du DoD (*Department of Defense*) a été formalisée par Bell et La Padula [BL74]. Elle est basée sur un modèle formel multiniveau permettant d'apporter la preuve que les règles de contrôle d'accès qu'elle définit satisfont effectivement les propriétés de sécurité exigées. Cette politique vise à préserver la confidentialité des informations.

D'autres modèles de politiques obligatoires ont été développées pour le maintien de l'intégrité. Parmi ces politiques, citons celle de Biba [Bib77] qui applique à l'intégrité un modèle multiniveau analogue à celui de Bell-LaPadula, et celle de Clark et Wilson [CW87] qui formalise des procédures en vigueur dans les systèmes commerciaux. Nous détaillons ces politiques dans le paragraphe suivant.

1.3 Modèles de sécurité

Les politiques d'autorisation sont généralement décrites par un modèle formel, ce qui permet de vérifier que la politique est complète et cohérente, et que la mise en œuvre par le système de protection est conforme.

1.3.1 Modèles basés sur les matrices d'accès

La notion de matrice de contrôle d'accès introduite par Lampson [Lam73] est un modèle simple et général. Elle est basée sur la notion de *sujets*, d'*objets* et de *droits*. Un *sujet* est une entité active s'exécutant pour le compte d'un utilisateur. Cette notion de sujet s'apparente très bien dans des systèmes tels qu'Unix à la notion de processus. Un *objet* est une entité considérée comme "passive", qui est définie par un état et des fonctions d'accès. Cette notion d'objet est à prendre au sens large : un sujet peut très bien avoir un droit sur un autre sujet : un processus p_1 peut avoir le droit de tuer un autre processus p_2. En ce sens, le processus p_2 sera aussi considéré comme un objet. À un instant donné, un sujet a un droit d'accès sur un objet si et seulement si le sujet est autorisé à exécuter la fonction d'accès correspondante sur cet objet. La protection consiste alors à définir un certain nombre de droits pour chaque sujet sur chaque objet et à garantir que seuls les accès correspondant à ces droits seront exécutés dans le système ; cet ensemble de droits est défini par les règles du schéma d'autorisation comme expliqué dans le paragraphe précédent.

L'ensemble des droits d'accès est souvent représenté dans une *matrice de droits d'accès*. Cette matrice représente la configuration courante de la protection du système. Les lignes représentent les sujets et les colonnes représentent les objets. L'intersection d'une ligne et d'une colonne contient l'ensemble des droits d'accès que possède le sujet correspondant sur l'objet correspondant. Ces droits d'accès constituent l'ensemble des opérations que le sujet peut réaliser sur l'objet.

Cette matrice de droits d'accès n'est pas figée. Elle évolue dans le temps en fonction de la création de nouveaux objets et de nouveaux sujets et en fonction des différentes opérations effectuées par les utilisateurs. Lorsqu'un nouveau processus est créé, une nouvelle ligne est insérée dans la matrice. Cette nouvelle ligne définit les droits d'accès accordés au processus ; ces droits sont initialisés en fonction de ceux de l'utilisateur pour le compte duquel il

s'exécute. De même, lorsqu'un nouvel objet est créé dans le système, une nouvelle colonne est insérée dans la matrice des droits d'accès. Elle représente les droits par défaut qu'ont les différents sujets sur cet objet. Ces "droits par défaut"sont également appelés *droits implicites* (par exemple, dans Unix, la commande *umask* permet de spécifier ces droits implicites). La composition de la matrice d'accès va également évoluer lors des différentes opérations effectuées par un utilisateur. Ces évolutions peuvent affecter ses propres droits mais aussi ceux des autres utilisateurs (dans le cas par exemple d'un accès à un objet en exclusion mutuelle). De même, un utilisateur peut délibérément demander à changer certains droits dans la matrice d'accès (sous Unix, un utilisateur peut demander à changer les droits d'accès des objets dont il est le propriétaire). Bien évidemment, ces modifications ne seront effectuées que si elles sont compatibles avec le schéma d'autorisation du système. En fait, la matrice de droits d'accès n'est jamais modifiée directement par un utilisateur mais seulement à travers des utilitaires du système qui effectuent les modifications requises si elles sont conformes au schéma d'autorisation.

Plusieurs modèles ont été basés sur la notion de matrice de contrôle d'accès. Nous en donnons ici deux exemples connus.

- Le modèle HRU (de Harrison, Ruzzo et Ullman) [HRU76] fut le premier modèle rigoureux bâti sur les notions énoncées ci-dessus. Il utilise une matrice d'accès et précise les commandes qui peuvent lui être appliquées.
- Le modèle Take-Grant [LS77] utilise un graphe et des règles de modification de ce graphe pour modéliser le contrôle d'accès sur des données. Chaque nœud du graphe représente un sujet ou un objet. Les arcs sont étiquetés et représentent des sous-ensembles d'un ensemble fini de droits. Ce modèle est bel et bien un modèle basé sur une matrice d'accès. En effet, le graphe représente l'état de protection du système, jouant le rôle de matrice. On peut d'ailleurs aisément construire cette matrice à partir des nœuds (qui constituent soit une ligne soit une colonne) et des étiquettes des arcs (qui constituent les droits à placer dans la matrice en fonction des nœuds que l'arc joint).

Le modèle HRU est une simple description des droits sous forme de matrice de contrôle d'accès, et ne permet pas de faire directement de vérification de cohérence ou de complétude. En revanche, le modèle Take-Grant permet de faire certaines vérifications de cohérence. Néanmoins, pour un système réel, de telles vérifications sont généralement coûteuses et incomplètes : on ne peut

vérifier, sans faire une énumération totale, qu'il n'est pas possible d'atteindre un état d'insécurité. Plus récemment, des extensions du modèle HRU ont été proposées pour résoudre ce problème, au prix de certaines restrictions sur les règles du schéma d'autorisation. C'est le cas en particulier du modèle TAM [San92] et du graphe des privilèges [Dac94].

1.3.2 Modèles basés sur la notion de treillis

Les exigences des politiques de sécurité militaires ou gouvernementales proviennent du fait que si une information est connue d'un ennemi, elle peut mettre en péril la sécurité nationale toute entière. Aussi, il est important, dans une telle approche, de représenter le coût que peut constituer la divulgation de chaque information dans le système. Ce coût est représenté par une classification. Ces classifications sont typiquement : NON-CLASSIFIÉ, CONFIDENTIEL, SECRET et TRÈS-SECRET et sont totalement ordonnées. De plus, est associé à chaque information un compartiment, constitué d'un ensemble de catégories définissant le domaine de l'information (telles que "nucléaire" ou "cryptographie" par exemple). Le niveau de sécurité d'une information comprend sa classification et son compartiment, c'est-à-dire les catégories auxquelles elle appartient. À chaque utilisateur est associé un niveau de sécurité représentant la confiance qui est accordée à cet utilisateur, définie par son habilitation et un compartiment (l'habilitation comme la classification peut prendre les valeurs NON-CLASSIFIÉ, CONFIDENTIEL, SECRET, TRÈS-SECRET).

Les niveaux de sécurité sont partiellement ordonnés selon la relation suivante : le niveau de sécurité d'une information défini par sa classification c et son compartiment C est dominé par le niveau de sécurité d'une autre information défini par sa classification c' et son compartiment C' si et seulement si $c \leq c'$ et $C \subseteq C'$. Nous représentons cette relation par la notation : $\preceq$. Elle permet de définir les flux d'information autorisés ou interdits dans le système (nous en donnons deux exemples dans la suite). Les niveaux de sécurité des utilisateurs sont partiellement ordonnés de la même façon.

L'exemple de Bell-LaPadula

Ce modèle présente un système sous forme d'une machine à états finis. Chaque état est défini comme une matrice ($SxO \rightarrow A$) qui pour chaque sujet $s \in S$ et chaque objet $o \in O$ décrit tous les accès autorisés $a \in A$. À chaque sujet est attribué un niveau de sécurité $h(s_i)$ et à chaque objet est attribué un niveau

de sécurité $c(o_j)$. Deux propriétés assurent qu'un état est sûr :

- La propriété simple : si $(s_i, o_j, Read) \in (S \text{x} O \text{x} A)$ alors $c(o_j) \preceq h(s_i)$.
- La propriété étoile : si $(s_i, o_j, Read) \in (S \text{x} O \text{x} A)$ et $(s_i, o_k, Write) \in (S \text{x} O \text{x} A)$, alors $c(o_j) \preceq c(o_k)$. Cette propriété vise à empêcher le flux d'information d'un niveau de sécurité donné vers un niveau de sécurité supérieur.

Ce modèle possède plusieurs inconvénients :

- L'information se dégrade constamment par surclassification. En effet, la propriété étoile impose que le niveau de sécurité d'une information ne peut qu'augmenter dans le système amenant donc peu à peu ce dernier dans un état où toutes les informations sont surclassifiées.
- Ce modèle n'empêche pas des canaux cachés de mémoire et des canaux cachés temporels (cf. §1.3.3).

De plus, McLean a montré qu'en utilisant ce modèle, il était possible de construire un système (qu'il appelle *System Z*) qui vérifie les deux propriétés précédentes mais qui est pourtant non sûr [McL85, McL87, LH89]. Ce système consiste simplement pour un utilisateur quelconque à effectuer une transition dans le système qui consiste à attribuer à tous les objets le niveau minimal. La réplique de Bell a été de déclarer que chaque niveau d'une information était fixe [Bel88]. On reviendra plus en détail sur le modèle de Bell-LaPadula dans le chapitre 3.

L'exemple de Biba

Le modèle de Biba est le modèle dual de Bell-LaPadula dans lequel il s'agit d'assurer l'intégrité d'un système. Chaque sujet et chaque objet se voient attribuer un niveau d'intégrité. Trois règles garantissent qu'un état du système est un état sûr :

- Un sujet ne peut *modifier* un objet que si le label d'intégrité du sujet domine le label d'intégrité de l'objet.
- Un sujet ne peut *observer* un objet que si le label d'intégrité du sujet est dominé par le label d'intégrité de l'objet.

- Un sujet A ne peut *invoquer* un sujet B que si le label d'intégrité du sujet B est dominé par le label d'intégrité du sujet A.

Ce modèle possède l'inconvénient dual du modèle de Bell et LaPadula au sens où l'intégrité de l'information ne peut que diminuer. Le risque est donc d'obtenir au bout d'un certain temps un système dans lequel toutes les informations ont un niveau d'intégrité faible. L'utilisation combinée de la politique de Bell-LaPadula et de celle de Biba peut amener à un système surprenant : toutes les informations sont classifiées au plus haut niveau et ne sont absolument pas intègres !

1.3.3 Les modèles de contrôle de flux

Les modèles basés sur la notion de flux d'information doivent leur apparition à la non-maîtrise des canaux cachés par les politiques de type Bell-LaPadula. Un **canal caché** désigne un chemin de communication non prévu ou non autorisé qui peut être utilisé pour transférer de l'information d'une manière violant la politique de sécurité du système d'information [ITS91]. On peut citer deux grandes familles de canaux cachés : les *canaux cachés de mémoire* et les *canaux cachés temporels*. Un canal caché de mémoire implique l'écriture directe ou indirecte d'un élément de stockage par un sujet et la lecture directe ou indirecte de ce même élément par un autre sujet d'un niveau inférieur. Les canaux de mémoire utilisent une ressource réelle (comme des secteurs sur disque), qui est partagée par deux sujets à différents niveaux de sécurité. On parle de canal caché temporel lorsqu'un processus module sa propre utilisation des ressources du système, afin de modifier le temps de réponse observé par un autre processus, lors de l'accès à ces mêmes ressources. Or un modèle de sécurité du type Bell-LaPadula contrôle uniquement les flux d'information par documents (fichiers) et donc ne fournit pas de solution à l'identification et l'élimination de canaux cachés. En fait, ce type de modèle gère le problème d'accès aux objets par les sujets mais en aucun cas celui du flux d'information par un sujet sans passer par l'intermédiaire d'objets identifiés. Les modèles de contrôle de flux d'information correspondent à une vue plus macroscopique et plus générale du système. On ne considère plus alors des opérations de lecture et d'écriture sur des fichiers mais des flux d'information entre sujets. Ces modèles permettent de spécifier les canaux de transmission d'informations, de préciser les canaux légitimes et d'identifier les canaux cachés. À titre d'exemple, on peut effectuer une analyse de contrôle de flux directement sur

chaque instruction du code d'un programme [Bry94].

1.3.4 La non-interférence

Le modèle de non-interférence [GM84] représente le système comme une machine à états définie par un ensemble U d'utilisateurs, un ensemble C de commandes permettant de changer l'état du système, un ensemble R de commandes de lecture du système, un ensemble O de sorties du système et un ensemble S d'états internes avec un état initial s_0. Une commande de l'ensemble R est donc une commande qui permet de visualiser les sorties du système sans le faire changer d'état alors qu'une commande de l'ensemble C est une commande qui permet de faire changer le système d'état sans observer de sorties.

La fonction de transition d'états est définie par :

$do : S \mathrm{x} U \mathrm{x} C \rightarrow S$
où $do(u, s, c)$ donne l'état résultant de l'exécution de la commande c effectuée par l'utilisateur u lorsque le système est dans l'état s.

La fonction de sortie est définie par :

$out : S \mathrm{x} U \mathrm{x} R \rightarrow O$
où $out(s, u, r)$ donne le résultat de la lecture (via la commande r) du système dans l'état s par l'utilisateur u.

L'historique d'un système est défini par la séquence $w = (u_1, c_1) \ldots (u_n, c_n)$, c'est-à-dire un ensemble de commandes $c_1, \ldots c_n \in C$ effectuées respectivement par les utilisateurs $u_1, \ldots u_n$ à partir de l'état s_0. Notons $\| w \|$ l'état dans lequel le système est arrivé à partir de l'état s_0 et l'exécution de la séquence w. La non-interférence exprime qu'un certain groupe d'utilisateurs G exécutant un certain ensemble de commandes de changement d'état A (A étant un sous-ensemble de C), n'interfère pas (c'est-à-dire ne peut être détecté) par un autre groupe d'utilisateurs G' exécutant un ensemble B (B étant un sous-ensemble de R) de commandes de lecture du système. On aura donc non-interférence, si pour chaque séquence w, chaque $v \in G'$ et chaque $r \in B$, on a :

$$out(\| w \|, v, r) = out(\| p_{G,A}(w) \|, v, r)$$

où $p_{G,A}$ est la séquence obtenue à partir de w en éliminant toutes les occurrences (u, c) où $u \in G$ et $c \in A$. Intuitivement, cela signifie que pour chaque v

de G' qui exécute une commande de lecture de B, les sorties du système sont pour lui identiques que les utilisateurs de G aient effectués des commandes de changement d'état de A ou pas.

Plusieurs raffinements de ce modèle ont été publiés. On peut citer notamment le modèle de *non déduction* [Sut86] et le modèle de *causalité* [d'A94], que nous décrivons plus en détail dans le §3.2.2.

Ce modèle est en fait très puissant car si on arrive à assurer la propriété de non-interférence dans un système, alors on est certain de contrôler tous les flux d'information et donc de maîtriser les canaux cachés. Il semble néanmoins qu'un tel modèle soit difficilement utilisable dans les systèmes réels. Il reste relativement difficile de passer du modèle mathématique à une véritable implémentation dans un système. Les travaux réalisés dans [Cal95] sont un des rares exemples d'implémentation d'une telle politique.

1.3.5 Modèles spécifiques

Certains modèles de sécurité correspondent à des exigences typiquement liées à un milieu particulier. Ainsi, le modèle de Clark et Wilson [CW87] vise à préserver l'intégrité des informations dans un système à caractère commercial (au sens de "système pour faire du commerce"). Il introduit deux notions couramment utilisées dans les systèmes commerciaux: les transactions bien formées et la séparation des privilèges. La première stipule que les utilisateurs ne peuvent manipuler les informations qu'au travers de procédures qui préservent l'intégrité des données (les *procédures de transformation*). La seconde impose que certaines opérations nécessitent l'intervention de plusieurs personnes pour être menées à bien (principe de la séparation des pouvoirs).

Le modèle de la muraille de Chine [BN89] étudie le problème de la confidentialité lié à l'intervention d'une personne dans différents milieux en conflit d'intérêt. Ainsi ce modèle définit la notion de *classe d'intérêt*. Il stipule qu'une personne ne peut pas être autorisée à consulter une information si elle connaît déjà une autre information qui est en conflit d'intérêt avec la première (c'est-à-dire dont les classes d'intérêt sont en conflit). Ainsi, par exemple, imaginons un consultant ayant accès aux informations de deux sociétés concurrentes A et B. Dès lors que le consultant a eu accès aux informations de la société A, il lui est interdit de consulter toute information de la société B puisque les classes d'intérêt de ces informations sont en conflit. En fait, le consultant a, au départ, la liberté de choix de consultation de l'information, mais chaque

décision qu'il prend dresse devant lui une barrière qu'il ne peut plus franchir. La muraille de Chine est une image de cette barrière. Cette politique correspond à une réglementation imposée aux agents de change britanniques car leur travail les amène à travailler pour différentes sociétés qui peuvent être en situation de conflit d'intérêt.

1.4 Evaluation de la sécurité

Après avoir présenté les différentes propriétés de sécurité que l'on peut désirer assurer dans un système ainsi que les principales familles de politiques de sécurité, on peut s'interroger sur l'évaluation de la sécurité des systèmes: comment peut-on dire qu'un système est *plus sûr* qu'un autre système?

1.4.1 Le livre orange

Le rapport *Trusted Computer Security Evaluation Criteria* [DoD85], plus connu sous le nom de TCSEC ou "livre orange", est devenu en 1985 une norme du Département de la Défense américaine (DoD). Ce document est la référence en matière d'évaluation de la sécurité des systèmes informatiques. Les critères offrent une classification à sept niveaux de sécurité, regroupés en quatre classes A,B,C,D (les niveaux sont A1,B1,B2,B3,C1,C2,D). Quatre familles de critères sont définies pour chaque niveau ; elles traitent respectivement de *la politique d'autorisation*, de *l'audit*, de *l'assurance* et de *la documentation*. L'évaluation d'un produit consiste à lui attribuer un des sept niveaux de sécurité. Cette attribution n'aboutit que si le produit répond à tous les critères du niveau en question. La politique d'autorisation distingue les deux types de politiques, discrétionnaire et obligatoire.

- Un système classé D est un système ne présentant aucune caractéristique particulière quant à la sécurité.
- Les critères exigent l'utilisation d'une politique discrétionnaire pour les niveaux C1 et C2.
- Les critères exigent l'utilisation d'une politique discrétionnaire **et** d'une politique obligatoire pour les niveaux B1, B2, B3.

- Un système A1 est fonctionnellement équivalent à un système B3 mais est caractérisé par l'utilisation de méthodes formelles de vérification qui permettent d'assurer que les contrôles discrétionnaires et obligatoires employés dans le système sont bien à même de protéger les informations sensibles manipulées par le système. Une documentation détaillée et complète est nécessaire pour démontrer que le système respecte bien les exigences de sécurité à tous les stades de la spécification, de la conception et du codage. Un exemple de système A1 est présenté dans [Wei92].

La politique obligatoire imposée par le livre orange est la politique de Bell-LaPadula (cf. §1.3.2). Comme nous l'avons dit, le fait que cette politique ne s'intéresse qu'au problème de confidentialité des données et qu'elle ne permette pas de prendre en compte les contraintes liées aux pratiques commerciales (présentées dans la politique de Clark et Wilson, cf. §1.3.5) a suscité la création de nouveaux critères dans d'autres pays. Nous citons ci-dessous l'exemple des ITSEC (*Information Technology Security Evaluation Criteria*) adoptés par la Communauté Européenne mais d'autres pays tels que le Canada avec les CTCPEC (*Canadian Trusted Computer Product Evaluation Criteria*) [CTC93] et le Japon [JCS92] ont également élaboré leurs propres critères d'évaluation.

1.4.2 Les ITSEC

Les ITSEC sont le résultat de l'harmonisation de travaux réalisés au sein de quatre pays européens : l'Allemagne, la France, les Pays-Bas et le Royaume-Uni [ITS91]. La différence essentielle que l'on peut noter entre le livre orange et les ITSEC est la distinction entre fonctionnalité et assurance. En effet, les ITSEC définissent un certain nombre de classes de fonctionnalité d'une part et un certain nombre de classes d'assurance d'autre part. Une classe de fonctionnalité décrit les mécanismes que doit mettre en œuvre un système pour être évalué à ce niveau de fonctionnalité. Une classe d'assurance permet, elle, de décrire l'ensemble des preuves qu'un système doit apporter pour montrer qu'il implémente réellement les fonctionnalités qu'il prétend assurer. Les classes d'assurance sont au nombre de six (E1 à E6). Parmi les classes de fonctionnalité, on retrouve les classes (F-C1, F-C2, F-B1, F-B2, F-B3) correspondant aux classes C1 à B3 définies dans les TCSEC. De plus amples détails sur les correspondances entre les ITSEC et les TCSEC peuvent être trouvés dans [BPB+90].

Les ITSEC utilisent le terme *cible d'évaluation* (*Target of Evaluation* ou TOE). Le contenu d'une TOE comprend : une politique de sécurité, une spécification des fonctions requises dédiées à la sécurité, une définition des mécanismes de sécurité requis et le niveau d'évaluation visé.

1.4.3 Les critères communs

Les critères communs (CC) [CC96] sont nés de la tentative d'harmonisation des critères canadiens (CTCPEC), des critères européens (ITSEC) et des critères américains (TCSEC). La version 1.0 des *Common Criteria for Information Security Evaluation* a ainsi été mise au point en janvier 96. Cette version est largement distribuée et a été soumise à commentaires jusqu'au mois de novembre 1996. Les objectifs des auteurs sont de parvenir à la mise au point d'une version 2.0 destinée à devenir une norme internationale.

Les CC contiennent deux parties bien séparées comme dans les ITSEC : fonctionnalité et assurance. De même que dans les ITSEC également, les CC définissent une TOE (*Target of Evaluation*) qui désigne le système ou le produit à évaluer, et la ST *Security Target* qui contient les objectifs de sécurité d'une TOE particulière et qui spécifie les fonctionnalités et les assurances offertes par la TOE afin de remplir ces objectifs. La *Security Target* pour une TOE représente la base d'entente entre développeurs et évaluateurs. Une *Security Target* peut contenir les exigences d'un ou de plusieurs *Protection Profiles* (PP) prédéfinis. Une des différences essentielles entre ITSEC et CC réside dans l'existence des *Protection Profiles*, qui avaient auparavant été introduits dans les Critères Fédéraux(*Federal Criteria*) [FC92]. Un *Protection Profile* définit un ensemble d'exigences de sécurité et d'objectifs, indépendants d'une quelconque implémentation, pour une catégorie de TOE. L'intérêt des *Protection Profiles* est double : un développeur peut inclure dans une *Security Target* un ou plusieurs *Protection Profiles*; un client désirant utiliser un système ou un produit peut également demander à ce que son système corresponde à un *Protection Profile* particulier, ceci lui évitant de donner une liste exhaustive de fonctionnalités et assurances qu'il exige du système ou produit. Une partie importante des CC est donc consacrée à la présentation détaillée de *Protection Profiles* prédéfinis.

Cette notion de PP représente la volonté américaine qui consiste à préférer évaluer des systèmes qui entrent dans le cadre de profils connus. Le cas d'un système ne cadrant pas exactement avec un profil connu conduit simplement

à l'élaboration d'un nouveau profil. La tendance européenne serait plutôt de définir systématiquement une TOE et ses exigences de sécurité et d'évaluer cette TOE sans nécessairement s'appuyer sur des profils prédéfinis. Les CC tentent de réaliser un compromis équitable entre ces deux positions.

1.4.4 Conclusion

Il n'est pas possible d'affirmer dans l'absolu qu'un système informatique est plus sûr qu'un autre système informatique : deux systèmes informatiques ne sont pas a priori destinés à être utilisés dans le même environnement, ne sont pas a priori soumis aux mêmes menaces. Le livre orange définit des critères permettant de classer des systèmes dans le cas bien particulier de systèmes militaires utilisant des politiques discrétionnaires et obligatoires. Cette classification ne sépare pas néanmoins la notion de critères fonctionnels (ce que fait le système) et les critères d'assurance (comment prouver que le système fait bien ce qu'il prétend faire). Les ITSEC proposent quant à eux de séparer ces deux types de critères et permettent donc d'affirmer que pour telle fonctionnalité précise, un système présente plus de garanties qu'un autre. Les critères communs viennent finalement tenter d'établir un compromis entre les différents critères existants et introduisent la notion de profil de protection sur lequel il est possible de se reposer pour construire un système sûr.

Notons enfin que l'ensemble de ces méthodes ne donnent qu'une version statique de la sécurité des systèmes et ne prennent donc pas en compte l'influence de l'utilisation de ces systèmes sur leur sécurité. Pour mieux prendre en compte cette influence, une méthode d'évaluation quantitative de la sécurité opérationnelle a été développée au LAAS [Dac94].

1.5 Le principe de la protection

Nous allons maintenant nous intéresser au problème spécifique de la protection dont nous rappelons que l'objectif est double : la protection vise à empêcher les opérations non-autorisées dans un système mais aussi à limiter la propagation des erreurs dans un système. Après avoir donné les définitions et principes relatifs à cette notion, nous tenterons d'expliquer les différents problèmes qui se posent dès lors que l'on désire assurer la protection de systèmes répartis. Nous finissons en mettant en avant les modifications à apporter à la vision classique de la protection dans le cas de systèmes composés d'objets répartis.

1.5.1 Sujets, objets et matrice de droits d'accès

L'autorisation est mise en œuvre par les mécanismes de **protection**. Ces mécanismes font classiquement intervenir les notions de *sujet*, d'*objet* et de *matrice de droits d'accès* que nous avons présentées dans le paragraphe 1.3.1.

Dans les systèmes réels, la matrice d'accès est en fait un peu différente de celle décrite précédemment. En effet, une matrice contenant une ligne par sujet du système et une colonne par objet du système serait très lourde à gérer ; aussi va-t-on réunir les sujets qui ont les mêmes droits sur les mêmes objets en *domaines de sujets* et utiliser une ligne de la matrice par domaine. Ces domaines sont eux-mêmes des objets, sur lesquels les sujets d'un domaine peuvent exécuter des opérations telles que changer les droits d'un domaine par exemple. Dans Unix, chaque vérification de droits d'accès pour un processus se fait uniquement sur l'utilisateur pour le compte duquel s'exécute le processus, ce qui revient à regrouper tous les processus d'un même utilisateur dans un même domaine de sujets. De même, Unix permet de regrouper les différents utilisateurs en *groupes* et ainsi d'attribuer des droits à ces groupes. Cette gestion des droits d'accès est bien sûr moins fine qu'une gestion par sujet du système mais elle est en général suffisante pour des systèmes dont la sécurité n'est pas un des objectifs principaux. En particulier, elle ne permet pas d'appliquer de façon optimale le principe du moindre privilège, selon lequel, lorsqu'un utilisateur effectue une opération autorisée dans le système, seuls les droits qui lui sont nécessaires pour réaliser cette opération doivent lui être accordés.

Une autre méthode pour réduire la taille de la matrice des droits d'accès consiste à regrouper dans des *domaines d'objets* les objets qui présentent les mêmes droits d'accès pour les mêmes sujets. La matrice ne contient plus alors qu'une colonne par domaine. Ces domaines sont en général exclusifs : quand un sujet entre dans un domaine, il perd tous les droits qu'il avait auparavant. Cette technique permet d'implémenter de façon simple les niveaux de classification des politiques obligatoires multiniveaux (cf. §1.3.2).

La matrice des droits d'accès n'est en réalité pas stockée sous la forme d'une matrice ; on préfère l'organiser de manière distribuée par lignes ou par colonnes.

- La gestion des droits d'accès par colonnes correspond à construire des listes de contrôles d'accès (*ACL* pour *access control list* en anglais). Une

liste de contrôle d'accès représente pour un objet o_j la liste des sujets s_i qui ont des droits d'accès pour o_j ainsi que la liste de ces droits d_{ij}. Cette représentation est typiquement celle d'Unix.

- La gestion des droits d'accès par lignes correspond à construire des listes de capacités (*capabilities* en anglais) [Fab74]. Une liste de capacités pour un sujet s_j est la liste des objets o_i pour lesquels le sujet s_j a des droits ainsi que l'ensemble de ces droits d_{ij}. Des expériences ont mis en œuvre des capacités par logiciel comme par exemple dans Hydra [WCC+74]. Des architectures de machines spécialisées ont également été développées, telles que l'IBM 38, l'IAPX 32 d'intel, le Plessey 250 [Kra80] ou le multi-processeur Cm* [SFS77]. Il est important de souligner que ces listes de capacités, associées à un sujet ne doivent en aucun cas être directement manipulables par le sujet en question. Il ne doit pas pouvoir étendre ses droits et surtout ne doit pas pouvoir extraire une des capacités pour la transmettre à un autre individu (ce qui pourrait être en contradiction avec la politique d'autorisation). C'est pourtant ce qui est réalisé dans le système Amœba d'Amsterdam [TMvR86] où les capacités sont considérées comme des tickets. On peut éviter le rejeu et la contrefaçon de capacités en insérant dans chacune d'elle l'identité de l'individu pour laquelle elle a été créée [Gon89]. Même si un utilisateur transmet une capacité à un autre utilisateur, ce dernier ne pourra l'utiliser car le système de protection pourra vérifier que cette capacité n'a pas été créée pour lui.

Ces deux notions, listes de contrôles d'accès et listes de capacités ne sont pas incompatibles mais sont souvent utilisées conjointement. Ainsi, dans Multics [Kra80], une liste de contrôles d'accès est associée à chaque segment et l'ensemble des droits que possède un processus sur un segment est copié dans le descriptif du processus et est vérifié par le matériel à l'exécution. Les listes de contrôles d'accès sont donc dans ce cas utilisées conjointement aux capacités.

1.5.2 Sous-système de sécurité, moniteur de référence et noyau de sécurité

Le livre orange a défini deux notions essentielles pour la construction de systèmes de sécurité : les notions de sous-système de sécurité ou base informatique de confiance (*Trusted Computing Base*) et moniteur de référence (*Reference Monitor*).

Un système réel complexe est construit à partir de multiples composants (matériels et logiciels). Certains de ces composants ne contribuent pas à satisfaire les objectifs de sécurité du système. D'autres, en revanche, sont destinés à satisfaire les objectifs de sécurité; ces derniers composants sont dits *dédiés* aux objectifs de sécurité. D'autres enfin, ne sont pas dédiés à la sécurité mais doivent cependant fonctionner correctement pour que le système puisse faire respecter la sécurité; ces composants sont dits *touchant* à la sécurité du système. La combinaison des composants dédiés à la sécurité et des composants touchant à la sécurité est appelée **sous-système de sécurité** ou **base de confiance (TCB)**.

On peut immédiatement se rendre compte que les mécanismes de contrôle d'accès réalisant la protection d'un système vont se trouver dans cette base de confiance. En ce qui concerne les composants non dédiés à la sécurité mais touchant à la sécurité, on peut citer entre autres les mécanismes d'identification. En effet, le fait de pouvoir se connecter à un système et donc de pouvoir utiliser ce système fait partie des fonctionnalités minimales qu'un système doit fournir, que ce système vise à être sécurisé ou non. Néanmoins, si pendant la phase de connexion d'un individu sur un système, l'identification de l'utilisateur n'est pas établie correctement, la sécurité du système est mise en péril. Aussi, des parties du système, comme la procédure de login par exemple sur Unix, qui n'est pas dédiée à la sécurité, doit faire partie de la base de confiance si l'on veut construire un système sûr.

Le **moniteur de référence** (cf. figure 1.1), tel qu'il est défini dans le livre orange, est la partie du système qui est chargée de valider tous les accès des sujets du système vers les objets du système. Le moniteur de référence doit obligatoirement posséder les trois propriétés suivantes :

- Le moniteur de référence doit être inviolable.
- Le moniteur de référence doit toujours être invoqué.
- Le moniteur de référence doit être vérifié correct.

La première propriété correspond à assurer l'isolation du moniteur de référence : il ne doit pas être possible de modifier le comportement du moniteur de référence lorsque le système est en fonctionnement. La seconde propriété signifie que le moniteur doit être incontournable : il ne doit pas être possible à un sujet d'accéder à un objet du système sans que l'accès ne soit contrôlé par

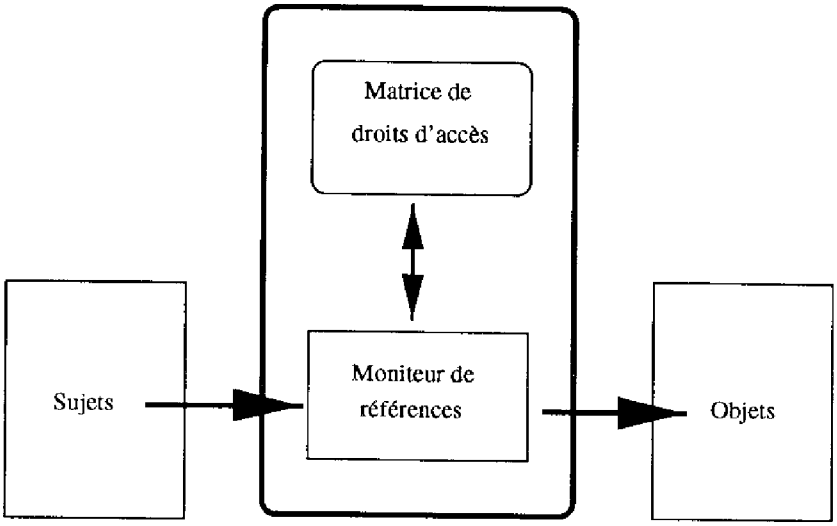


FIG. 1.1 – *Moniteur de référence*

le moniteur de référence. La troisième propriété enfin signifie que le moniteur de référence doit être suffisamment simple pour être analysé et vérifié : on doit pouvoir s'assurer que le moniteur de référence *fait bien ce qu'il est supposé faire* et rien d'autre. Si les deux premières propriétés semblent plutôt relever de l'utilisation d'un matériel adéquat, la troisième propriété relève quant à elle de l'utilisation d'un modèle formel sous-jacent.

Landwehr dans [Lan83] et le livre orange définissent le **noyau de sécurité** comme l'ensemble des moyens matériels et logiciels qui permettent de réaliser un moniteur de référence. Nous utiliserons dans la suite de ce mémoire une définition légèrement différente du noyau de sécurité. Nous considérerons le noyau de sécurité de façon un peu plus générale: le noyau de sécurité est un ensemble de moyens matériels et logiciels dédiés à la sécurité du système et autour duquel sont construites toutes les fonctions du système touchant à la sécurité. Ce noyau est suffisamment simple et petit pour être analysé et prouvé correct ; il est également incontournable et inviolable. Cette notion est un peu plus générale que la notion explicitée par Landwehr au sens où un noyau de sécurité n'implémente pas forcément le moniteur de référence, qui suppose la gestion d'un certain nombre de structures de données dont la matrice des droits d'accès par exemple.

De tels noyaux de sécurité ont déjà été développés. Ainsi par exemple, le micro-noyau TMach [BTM88] implémente des règles qui permettent de contrôler tous les accès à tous les objets locaux qu'il gère. Ces objets sont en l'occurrence des *portes* et des *tâches*. De même, le noyau de sécurité Scomp [Fra83] implémente ce type de contrôles. Ce noyau est développé à l'aide d'un matériel spécifique et présente une interface permettant la construction de systèmes sûrs.

Nous allons maintenant aborder les différents problèmes que l'on est amené à résoudre dès lors que l'on désire sécuriser des systèmes répartis.

1.6 La protection dans les systèmes répartis

Un système réparti (ou distribué) peut se définir comme un ensemble d'éléments de traitement (processeurs et mémoires) reliés par des organes de communication qui leur permettent d'échanger de l'information. Sécuriser un système réparti devient alors une tâche très difficile car il n'existe pas d'état global [Ray87] d'un tel système. Autrement dit, il n'est pas possible à un instant donné de savoir exactement *qui fait quoi* dans un système réparti. Si on peut assurer de façon indépendante la sécurité de chacun des sites pris isolément, il est impossible de garantir la sécurité du système global : comment être sûr, à un instant donné, que tous les sites qui composent le système sont sûrs et que les moyens de communication sont fiables? Dès lors, il apparaît clairement que le point essentiel de la sécurité dans les systèmes répartis va être lié à un problème de confiance. En qui ou en quoi peut-on avoir confiance à un instant donné? Étant donnée cette confiance, comment va-t-on assurer la protection du système?

1.6.1 Protection centralisée

La protection telle que nous venons de la présenter, basée sur une notion de TCB, semble intuitivement imposer une gestion centralisée de la sécurité. Assurer une gestion centralisée de la protection consiste à placer la confiance en une machine qui va être un passage obligé pour tous les accès des sujets du système réparti aux objets répartis. Cette machine joue en fait le rôle de moniteur de référence dont nous avons parlé précédemment : elle doit être un médiateur incontournable de toutes les interactions sujets-objets. Cette

structure centralisée a été effectivement adoptée par certains projets de systèmes répartis, dont le serveur de fichiers sécurisé de la Newcastle Connection [RR83]. Dans ce projet, les utilisateurs depuis leur terminaux, n'ont accès à leurs fichiers que par l'intermédiaire d'un *gestionnaire de fichiers sécurisé* centralisé dont le rôle est de gérer et de vérifier l'ensemble des droits d'accès.

L'avantage d'une telle approche est qu'elle permet d'avoir une politique d'autorisation cohérente et facile à maintenir : la politique d'autorisation est gérée sur une seule machine pour l'ensemble du système. Mais cette approche centralisée est critiquable dans le sens où la sécurité du système réparti repose entièrement sur une seule machine, qui devient donc un point dur du système (*single point of failure* en anglais). Si cette machine subit une défaillance due à des fautes accidentelles, la sécurité du système entier est mise en défaut. Cette machine devient également la cible privilégiée pour d'éventuelles intrusions dans le système. Un individu ayant réussi à se rendre maître du gestionnaire de fichiers de la Newcastle Connection devient libre de réaliser n'importe quelle action dans le système (il peut à sa guise, se donner des droits et modifier les règles de la politique d'autorisation). Une telle approche centralisée peut être également critiquable du point de vue des performances. En effet, toutes les interactions sujets-objets devant obligatoirement être contrôlées par une même machine, cette dernière risque de devenir un goulot d'étranglement, d'autant plus que le nombre de machines du système distribué est important.

1.6.2 L'approche du livre rouge

Une autre approche est celle adoptée par le NCSC *National Computer Security Center* dans le livre rouge [TNI87]. Ce rapport est en fait une adaptation aux systèmes répartis des critères d'évaluation du livre orange. Dans cette approche, chacun des sites qui composent le système possède une TCB et par conséquent peut être évalué en fonction des critères du livre orange. Chaque TCB est chargée de contrôler tous les accès locaux mais aussi les accès distants. Ce qu'il est important de noter est que dans cette approche, chaque TCB fait confiance aux autres TCB. Si un sujet s_1 accède à un objet distant o_2, la requête est transmise par la TCB du site 1 à la TCB du site 2. Cette dernière fait confiance à l'authentification de l'utilisateur par la TCB du site 1 et va donc consulter les listes de contrôles d'accès de l'objet o_2 pour autoriser ou refuser l'accès demandé. De même, la TCB du site 1 fait confiance la vérification des droits du sujet s_1 par la TCB du site 2. L'ensemble des TCB du système qui se font mutuellement confiance est appelé *Network Trusted*

Computing Base (ou NTCB). La sécurité globale du système repose donc sur la sécurité de cette NTCB (cf. figure 1.2).

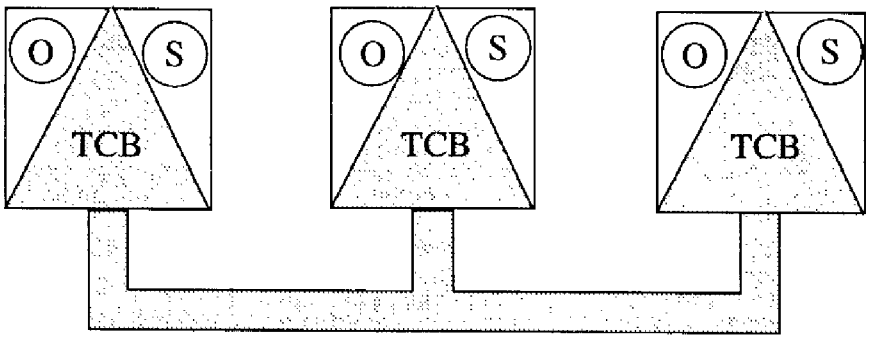


FIG. 1.2 - *L'approche du livre rouge*

Cette gestion décentralisée des informations peut poser des problèmes de cohérence. En effet, la matrice de droits d'accès est répartie sur tous les sites du réseau et il semble difficile de gérer une politique d'autorisation cohérente dans une telle situation (si chaque site n'a besoin de connaître que ses objets locaux, il doit en revanche connaître forcément tous les sujets du système global pour être capable de vérifier les accès qu'ils pourraient réaliser localement). Une des faiblesses de cette approche est également la confiance qui doit être accordée à tous les administrateurs de sécurité de chacun des sites. En effet, chaque TCB est considérée incontournable et inviolable, donc il n'est pas possible à un intrus de contourner les mécanismes de protection de chacun des sites pris isolément. En revanche, chaque TCB est bien administrée localement par une personne que l'on juge digne de confiance (pour maintenir la matrice de droits d'accès et les règles de la politique d'autorisation). Si un administrateur d'une TCB quelconque est corrompu, l'ensemble du système est corrompu puisque les autres TCB lui font "aveuglément" confiance[1]. Enfin, cette approche semble ne pas être compatible avec les systèmes ouverts d'aujourd'hui. En effet, avec le développement actuel des réseaux comme Internet, il devient tout à fait possible à une personne de connecter depuis son domicile son ordinateur personnel au système. Comment alors peut-on être sûr de la sécurité de cette machine puisque son possesseur en est totalement

1. Cette confiance "aveugle" peut être limitée, en considérant des niveaux d'habilitation correspondant à la confiance que l'on a dans chaque machine. Ainsi, des informations d'un haut niveau de classification ne seront jamais transmises à des machines de bas niveau d'habilitation.

maître et seul administrateur par définition? De plus, comment dans ce cas, assurer une gestion cohérente de la politique d'autorisation?

1.6.3 Distribution des fonctions de sécurité: l'approche Kerberos

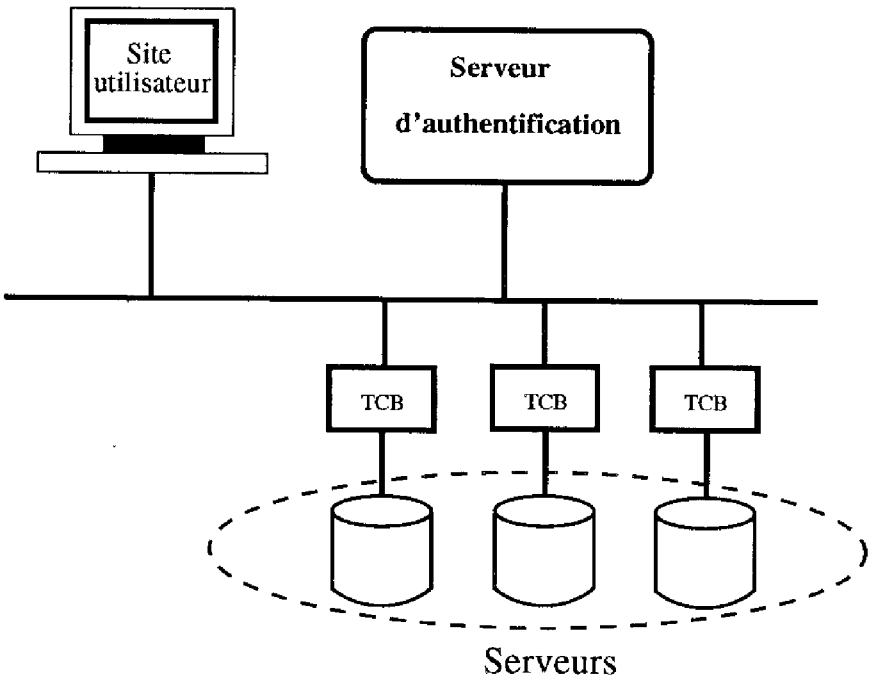


FIG. 1.3 – *L'approche Kerberos*

Le serveur d'authentification Kerberos [KN93] développé par le MIT dans le cadre du projet Athena vise à distribuer les fonctions de sécurité sur des sites de confiance. Ainsi, dans le projet Athena, l'authentification est gérée par un serveur d'authentification centralisé (Kerberos) alors que l'autorisation est gérée indépendamment par chacun des sites du système. Le serveur Kerberos doit bien sûr être digne de confiance pour les autres serveurs. Tout client voulant accéder à des informations sur un des serveurs devra obligatoirement se faire authentifier par le serveur d'authentification et pourra ensuite accéder aux informations sur le serveur (cf. figure 1.3). Ce serveur vérifiera que l'utilisateur a bien été authentifié par le serveur d'authentification dans lequel

il a confiance. Ensuite, il décidera lui-même d'accepter ou de refuser l'accès que le client a souhaité. Si dans cette approche, il n'est pas nécessaire de faire confiance aux sites utilisateurs (comme dans l'approche du livre rouge), il devient difficile de gérer une politique d'autorisation cohérente (chaque serveur décide de sa propre politique d'autorisation) et le serveur d'authentification reste un point dur du système[2]. La même approche a également été adoptée dans SESAME [Par91]. Dans SESAME, chaque utilisateur se fait authentifier par un serveur d'authentification centralisé et obtient un *Privilege Attribute Certificate* (PAC) qu'il peut ensuite présenter aux serveurs auxquels il désire accéder. De même que dans Kerberos, chaque serveur est cependant responsable de la protection des informations qu'il gère et doit donc pour cela, posséder sa propre TCB. Il peut ainsi décider en fonction du PAC présenté si son possesseur est autorisé ou non à accéder aux informations.

1.6.4 L'approche Delta-4

Cette approche [BD90] consiste à ne faire aucune confiance à aucun site particulier, mais seulement à un quorum de sites. Ainsi, un utilisateur ne sera pas authentifié seulement par un site, mais au contraire par une majorité de sites dits de sécurité, pour pouvoir par la suite obtenir des accès à des serveurs distants. Ainsi dans cette approche, on supporte la défaillance d'une *minorité* de sites de sécurité sans mettre en péril la sécurité du système tout entier [Bla92]. Pour cela, est appliquée à ce serveur, une technique de tolérance aux fautes accidentelles particulière : la Fragmentation-Redondance-Dissémination [DBF91] [FP85].

La fragmentation consiste à découper des données en fragments de telle manière que l'information contenue dans un ou plusieurs fragments soit non-significative. On va ainsi tolérer des intrusions contre la confidentialité sur un nombre limité de fragments. La dissémination consiste à répartir les fragments dans le système de manière à isoler les fragments les uns des autres. La redondance peut-être introduite pendant la phase de fragmentation ou lors de la dissémination. La Fragmentation-Redondance-Dissémination pose deux

2. Le serveur d'authentification est un point dur au sens où un intrus qui s'en rendrait maître pourrait se faire passer pour n'importe qui. En revanche, ce n'est pas nécessairement un point dur pour la disponibilité, puisque le serveur Kerberos peut être composé d'un serveur "maître"et d'un ou plusieurs serveurs "esclaves", chacun pouvant servir à l'authentification.

problèmes à un intrus tentant une attaque contre la confidentialité :

- d'une part, obtenir suffisamment de fragments pour disposer d'une partie suffisante de l'information ;
- d'autre part, éventuellement, réordonner les fragments, c'est-à-dire diminuer l'entropie (le désordre) de l'ensemble des fragments récupérés; ce désordre peut être dû à une fonction cryptographique.

La première application de la Fragmentation-Redondance-Dissémination a été réalisée sur un système réparti de stockage de fichiers persistants. Elle consiste simplement à fragmenter un fichier sur un site utilisateur et à disséminer ces fragments sur un ensemble de sites particuliers appelés sites de stockage. La figure 1.4 montre les différentes étapes de l'écriture de fichiers persistants.

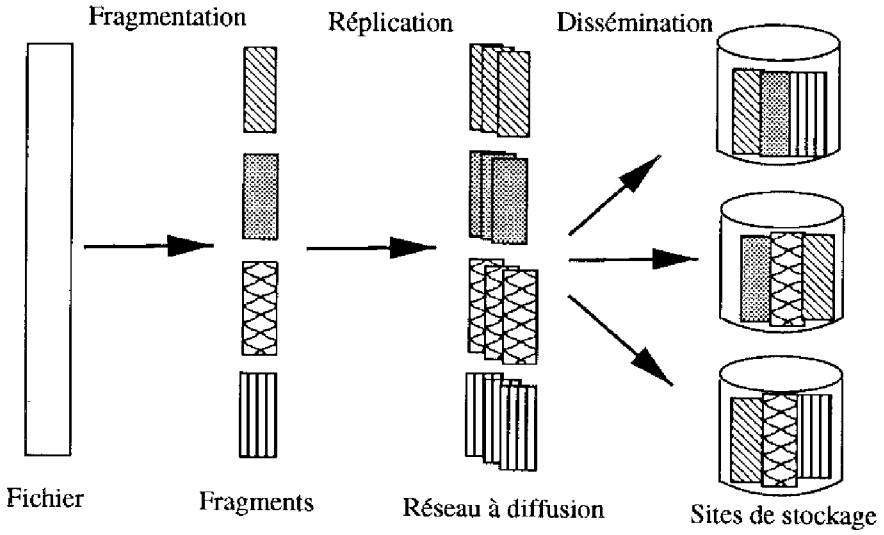


FIG. 1.4 - *Fragmentation-Redondance-Dissémination appliquée au stockage de fichiers*

La disponibilité sera assurée par la redondance, tandis que pour l'intégrité on utilisera une technique classique de signature cryptographique ou des fonctions de compression appliquées à l'information non-fragmentée ou aux fragments. Cette méthode est particulièrement adaptée aux systèmes répartis.

En effet, l'approche Fragmentation-Redondance-Dissémination s'appuie sur la notion de répartition géographique pour améliorer la sécurité: l'intrusion d'une partie du système réparti ne donne accès qu'à des fragments isolés, ne contenant donc pas d'information significative.

Dans Delta-4, le serveur de sécurité est composé de plusieurs sites de sécurité (cf. figure 1.5). Chacun d'eux a les mêmes fonctionnalités, mais pas obligatoirement les mêmes données. Cependant, pour les sites utilisateurs comme pour les autres serveurs, aucun site de sécurité ne se distingue des autres. Chaque site de sécurité est administré par un administrateur de sécurité différent et l'administrateur d'un site particulier n'a aucun droit sur les autres sites de sécurité. Cette séparation de pouvoirs évite le problème classique des systèmes distribués dans lesquels un seul administrateur de sécurité a tous les pouvoirs sur toutes les machines. Dans cette approche, il est possible de tolérer les intrusions que les administrateurs seraient susceptibles d'effectuer par abus de pouvoir.

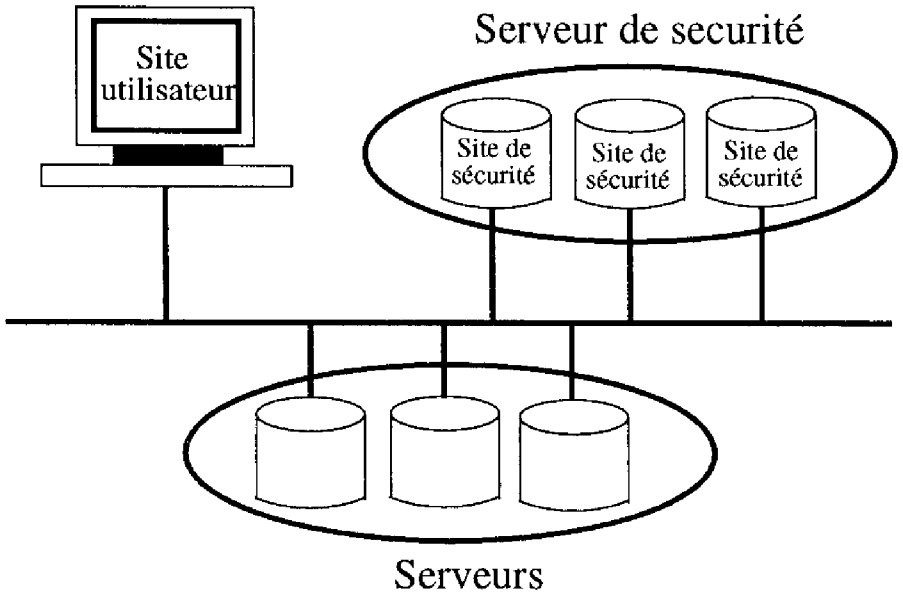


FIG. 1.5 – *L'approche Delta-4*

Les fonctionnalités de sécurité mises en œuvre au niveau des serveurs de sécurité assurent que la défaillance d'un site de sécurité n'a aucune répercussion sur la sécurité globale du système. D'autre part, il est fait l'hypothèse qu'il

existe toujours une majorité non-défaillante de sites de sécurité : la confiance dans un site ou dans une minorité de sites est limitée, mais on peut avoir confiance dans l'ensemble des sites de sécurité. La probabilité que plus d'une intrusion ou d'une faute accidentelle survienne simultanément est considérée faible, et la probabilité qu'une majorité de sites de sécurité soient défaillants en même temps est considérée pratiquement négligeable.

L'hypothèse de confiance dans l'ensemble des sites de sécurité signifie qu'à la différence des approches traditionnelles, il n'existe de TCB sur aucun site. On a confiance dans l'ensemble des sites, ou plutôt, dans une majorité d'entre eux.

Les moyens permettant d'assurer la sécurité du serveur de sécurité sont les suivants :

- Les sites de sécurité sont des sites spécialisés sur lesquels les utilisateurs ne peuvent se connecter. Seul, l'administrateur d'un site de sécurité peut se connecter sur ce site. De plus, les utilisateurs n'ont accès aux serveurs de sécurité qu'au travers de fonctions précises prédéfinies.
- La technique de tolérance aux fautes accidentelles et intentionnelles consiste à répliquer les traitements sur tous les sites de sécurité pour assurer la disponibilité et l'intégrité, à répliquer les données non-confidentielles pour assurer la disponibilité et l'intégrité, à fragmenter les données confidentielles pour en assurer la confidentialité, l'intégrité et la disponibilité et à mettre en cohérence les traitements et les données.

Soulignons, que dans cette approche, le serveur de sécurité est responsable de l'authentification des utilisateurs du système réparti mais aussi de la gestion des droits des ces utilisateurs. L'autorisation d'effectuer une opération dans le système est prise en charge par le serveur de sécurité.

1.7 La protection dans les systèmes d'objets répartis

Dans cette partie, nous commençons par présenter les concepts de base de la programmation orientée objets puis nous donnons un aperçu de la norme CORBA en focalisant plus particulièrement sur les aspects liés à la sécurité.

1.7.1 Conception et programmation par objets : définition et principes

La majorité des définitions et principes que nous énonçons dans ce chapitre sont extraites du livre de Bertrand Meyer [Mey88a], l'inventeur du Langage *Eiffel*.

La conception par objets repose sur une idée apparemment élémentaire : les systèmes informatiques réalisent certaines actions sur certains objets. Pour obtenir des systèmes souples et utilisables, il vaut mieux structurer le logiciel autour des objets plutôt qu'autour des actions. Cette nouvelle méthode de conception de programme assez récente a conduit à l'élaboration de nouveaux langages de programmation dits "*langages orientés objets*" tels que *Smalltalk* [GR83], *C++* [Str92], *Eiffel* [Mey88b]. Ces langages, même s'ils diffèrent par certains points, sont tous bâtis à l'aide des notions essentielles que sont : les *types abstraits*, les *classes*, les *objets*, les *méthodes*.

La conception par objets repose sur une description du système à l'aide de types abstraits. Une spécification de type abstrait décrit un ensemble de structures de données non pas par son implémentation, mais par une liste de fonctions disponibles sur les structures de données et par les propriétés formelles de ces fonctions. Ainsi, une pile sera par exemple vue comme une structure pour laquelle les fonctions disponibles sont : empiler un nouvel élément, enlever l'élément du sommet, tester si la pile est vide, etc.

Une *classe* est le terme technique qui s'applique aux langages à objets pour décrire des ensembles de structures de données ayant des propriétés communes. Par rapport aux types abstraits, une classe correspond à l'implémentation d'un type de données abstrait, pas le type de données lui-même. Par exemple, une implémentation du type abstrait *Pile* peut reposer sur un tableau. Cette implémentation basée sur un tableau correspondra à une classe. Chaque classe offre un certain nombre de fonctions (correspondant aux fonctions définies par le type abstrait correspondant). Ces fonctions sont également appelées *méthodes*. Enfin, il est nécessaire de définir les relations importantes qu'il peut exister entre les classes. Plus précisément, deux relations doivent être mentionnées : *client* et *descendant* :

- Une classe est cliente d'une autre si elle utilise les méthodes de cette autre classe, définies dans l'interface. Par exemple, comme nous l'avons dit plus haut, nous pouvons implémenter le type de données abstrait

Pile à l'aide d'un tableau. La classe PILE correspondante devient alors de ce fait cliente de la classe TABLEAU.

- Une classe est descendante d'une ou plusieurs autres classes si elle est une extension ou une spécialisation de ces classes. Une classe descendante de la classe C est aussi appelée *sous-classe* de C. C'est la notion d'*héritage*, qui peut être *simple* ou *multiple*. Ainsi par exemple, supposons que l'on désire créer une bibliothèque graphique. L'une des classes pourrait décrire des polygones avec des opérations telles que le calcul du périmètre, la translation, la rotation, etc. Nous pouvons ensuite vouloir définir une nouvelle classe qui représente les rectangles. Nous pouvons bien sûr la construire à partir de rien. Mais un rectangle appartient à la famille des polygones ; beaucoup de ces méthodes sont donc les mêmes que celles de la classe POLYGONE. Aussi, nous pouvons définir la classe RECTANGLE qui *hérite* de la classe POLYGONE. Cela signifie que toutes les méthodes de POLYGONE (que l'on nomme *classe parent*) sont applicables de la même façon à la classe héritière. Par ailleurs, il est vrai que certaines versions spéciales de certaines opérations sont spécifiques d'un rectangle (il y a une meilleure façon de calculer le périmètre d'un rectangle que pour un polygone quelconque). Aussi sera-t-il possible de redéfinir certaines des méthodes du parent de façon à ce qu'elles aient une implémentation différente. Cette propriété est appelée *surcharge de méthodes.*

Un *objet* est défini comme une *instance* d'une classe. Il n'y a pas d'autre objet que des instances de classes. Il est important de bien distinguer la différence entre objets et classes : les objets existent seulement pendant l'exécution ; les classes sont des descriptions purement statiques d'ensembles possibles d'objets, qui sont les instances des classes. À l'exécution, il n'y a que des objets. Dans un programme écrit dans un langage orienté objet, nous ne voyons que des classes. Au niveau de base, les objets sont des enregistrements, comme dans des programmes en Pascal ou C. Comme un enregistrement, un objet occupe de la place mémoire pendant l'exécution, et comporte des zones appelées *champs*. Ces champs peuvent correspondre à des données élémentaires (entiers, caractères) mais aussi à des références sur d'autres objets. C'est ainsi que la notion d'objets partagés est introduite.

1.7.2 CORBA

1.7.2.1 Introduction

CORBA (*Common Object Request Broker Architecture*) est une norme pour les systèmes à objets distribués développée par l'OMG (*Object Management Group*). L'OMG est un consortium de vendeurs composé de HP, IBM, AT&T, Sun/SunSoft, Iona, Expersoft, Novell, Microsoft, BNR, Groupe Bull, ICL, Olivetti, Siemens, Objectivity, Ontos, Versant et beaucoup d'autres, le but de ce consortium étant le développement d'une norme garantissant l'interopérabilité de systèmes basés sur les objets. Les différentes compagnies membres de l'OMG développent des produits commerciaux qui supportent ces normes et d'autres développent des applications qui utilisent ces normes.

CORBA fournit des mécanismes grâce auxquels les objets peuvent envoyer des requêtes et recevoir des réponses de façon transparente. Ces communications se font grâce à un ORB (*Object Request Broker*). CORBA définit les interfaces de programmation pour accéder à l'ORB (cf. figure 1.6).

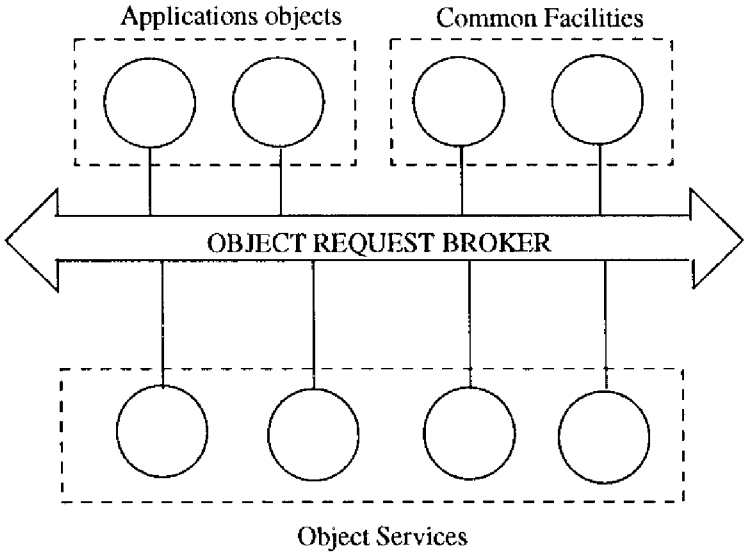


FIG. 1.6 – *CORBA*

– L'ensemble appelé *Object Services* représente une collection de fonctions de bas-niveau destinées à utiliser et implémenter les objets.

- Les *Common Facilities* sont un ensemble de fonctionnalités utiles à beaucoup d'applications.
- Les *Application Objects* constituent l'ensemble des applications spécifiques des utilisateurs qui utilisent le système.

En ce qui concerne la sécurité de tels systèmes, l'OMG n'a pas encore à proprement parler défini de normes mais a rédigé un certain nombres de rapports dressant les grandes lignes de ce que doit être la sécurité des systèmes à objets [OMG94b, OMG94a, OMG95]. Nous en dégageons ici les grands principes.

Plusieurs types de menaces contre de tels systèmes sont envisagés par l'OMG :

- Un objet ne doit pas pouvoir se faire passer pour un autre objet et ainsi pouvoir réaliser dans le système l'ensemble des opérations que l'objet dont il a pris l'identité peut réaliser.
- Un objet ne doit pas pouvoir consulter de l'information qu'il n'est pas autorisé à consulter (menace directement liée à la confidentialité).
- Un objet ne doit pas pouvoir modifier, créer ou détruire des informations de manière illégitime.
- Les communications doivent être protégées de façon à ce qu'on ne puisse lire des informations confidentielles mais aussi de façon à ce qu'on ne puisse modifier ou détruire des informations.

Pour faire face à cet ensemble de menaces, un système à objets doit fournir les fonctions de sécurité suivantes : authentification des objets, contrôles d'accès au niveau des objets, intégrité et confidentialité des communications. L'OMG ajoute à ces fonctions la *non-répudiation*, notion que nous allons expliquer dans les paragraphes suivants. Il est évident qu'un système ne doit pas forcément fournir toutes ces fonctions de sécurité : chaque fonction est propre à répondre à une ou plusieurs menaces considérées. Il faut tout d'abord définir les menaces contre lesquelles on veut agir et en conséquence implanter dans le système la fonction adéquate.

1.7.2.2 Identification et authentification

CORBA définit un "principal" comme une personne ou un objet enregistré dans le système et qui est authentifiable par le système. Dans le cas de per-

sonnes, cette authentification peut utiliser des mécanismes classiques tels que mots de passe ou des mécanismes basés sur des cartes à puces (par exemple). Les objets quant à eux peuvent se faire authentifier grâce à des mécanismes utilisant des clés cryptographiques.

Un principal qui est à l'origine d'une opération dans le système a au moins une et éventuellement plusieurs identités qui sont utilisées pour :

- enregistrer les actions qu'il réalise (*audit identity*),
- obtenir des droits sur des objets (*access identity*),
- l'identifier comme émetteur d'un message,
- l'identifier comme consommateur de ressources du système.

Un principal authentifié peut acquérir des privilèges qui lui seront nécessaires pour être autorisé à accéder aux objets du système. Ces privilèges (*privileges attributes*) peuvent contenir outre une identité, des rôles, des groupes, une habilitation (dans le cas de politique obligatoire) ou des capacités. Les privilèges d'un principal, associés aux informations concernant son identité (*identity attributes*) constituent un *credential* (une pièce justificative d'identité).

1.7.2.3 Autorisation et contrôle d'accès

Deux niveaux de contrôles d'accès sont définis dans CORBA. Lors de l'invocation d'un objet, un contrôle d'accès est automatiquement effectué par le système suivant une politique globale (*object invocation access policy*). De plus, chaque objet peut décider de réaliser en plus ses propres contrôles d'accès suivant sa propre politique (*application access policy*).

CORBA n'impose aucune politique d'autorisation globale particulière. Il est précisé que cette politique peut être discrétionnaire ou obligatoire. L'utilisation de listes de contrôles d'accès, de capacités, de labels doit être possible. Le contrôle d'accès s'effectue en fonction des privilèges de l'objet client (privilèges qui lui ont été délivrés lors de l'authentification) et des attributs de contrôle de l'objet invoqué (*control attributes*). Ces attributs de contrôle peuvent être des listes de contrôle d'accès ou des labels (dans le cas de politique obligatoire, le label de l'objet invoqué est comparé à celui de l'objet client en fonction des règles obligatoires de la politique de sécurité). Les attributs de contrôle d'un objet lui sont affectés lors de sa création.

1.7.2.4 Protection des communications

Les différents messages échangés par les objets doivent pouvoir être protégés de façon à assurer la confidentialité et/ou l'intégrité de ces messages. Ceci implique l'utilisation de mécanismes de signature (pour l'intégrité) et de chiffrement (pour la confidentialité). Deux objets peuvent en outre établir ce que CORBA nomme une association sûre *secure association*. Cette association consiste d'une part pour les deux objets à établir une confiance réciproque en s'authentifiant mutuellement et d'autre part à décider conjointement d'un mode de protection pour les messages qu'ils vont échanger.

1.7.2.5 Délégation

Dans un système composé d'objets, lorsqu'un client demande à un objet de réaliser une opération, l'objet peut demander à d'autres objets de réaliser des opérations pour lui et ainsi de suite. Cet enchaînement des opérations au sein des objets est pris en compte par CORBA. Un objet qui demande à un autre objet de réaliser une opération pour lui doit pouvoir lui transmettre une partie de ses droits. Cette délégation doit être contrôlée de façon à ce que l'objet à qui on délègue des droits ne puissent les utiliser pour une autre opération que celle pour laquelle ils ont été accordés. Plusieurs schémas de délégation sont définis dans CORBA. Prenons l'exemple d'un objet O désirant invoquer l'objet O' par l'intermédiaire de l'objet I :

- La *délégation simple* de droits signifie que l'objet O délègue une partie de ses droits à l'objet I, et que l'objet I va utiliser uniquement cette délégation lorsqu'il accède à l'objet O' pour O.
- La *délégation composite* de droits signifie que l'objet O délègue une partie de ses droits à l'objet I, et que l'objet I peut utiliser à la fois ses propres privilèges et la délégation reçue lorsqu'il accède à l'objet O'. L'objet O' reçoit ces deux ensembles de privilèges séparément.
- La *délégation combinée* de droits signifie que l'objet O délègue une partie de ses droits à l'objet I, et que l'objet I va combiner cette délégation avec ses propres privilèges sous la forme d'un seul *credential*. L'objet O' ne peut alors distinguer quels privilèges proviennent de quel objet.

Notons que dans tous ces cas, l'objet I peut de nouveau déléguer les droits reçus de l'objet O à d'autre objets.

1.7.2.6 La non-répudiation

La propriété de non-répudiation garantit qu'un objet ayant réalisé une action dans le système ne puisse nier l'avoir réalisée. Les contrôles de non-répudiation ne sont pas automatiquement effectués à chaque appel de méthode, mais c'est l'application elle-même qui doit explicitement effectuer ces contrôles si elle en éprouve le besoin. CORBA définit la notion d'*evidence* qui constitue la preuve qu'une action a été réalisée. Les deux types de preuves qui sont plus particulièrement mentionnés dans CORBA sont la preuve de création de message et la preuve de réception de message. Les *Common Facilities* fournissent donc, au travers de leur interface, des fonctions qui permettent de créer une preuve qu'un message a bien été créé dans le système ou une preuve qu'il a bien été reçu. Ainsi, l'émetteur de message ne peut nier l'avoir émis et le récepteur d'un message ne peut nier l'avoir reçu. Ces preuves sont générées par le *Delivery Authority* et vérifiées par l'*Evidence Generation and Verification*.

1.7.2.7 Conclusion sur CORBA

Le nombre élevé de participants à l'OMG et le poids des intérêts économiques que chacun des partenaires essaie de préserver fait que l'ensemble des propriétés que nous venons d'énoncer restent relativement vagues. Un certain nombre de points n'ont pas été clairement explicités dans CORBA. En effet, par exemple, il n'est pas dit explicitement dans CORBA si tous les sites du système distribué considéré sont considérés comme identiques ou si certains seront privilégiés (comme dans l'approche Delta-4 ou Kerberos par exemple). De même, il reste à déterminer quelles fonctions de sécurité seront présentes sur quelles machines et sous quelle forme (utilisation de TCB locale, de noyau de sécurité, etc). Puisque l'OMG n'explicite pas de façon détaillée l'ensemble des mécanismes à mettre en œuvre pour sécuriser des systèmes à objets, diverses propositions de systèmes *conformes* à l'architecture CORBA ont depuis été présentées [DBWL95].

1.8 Conclusion

Les approches actuelles, qu'elles soient centralisées (du type Newcastle Connection) ou distribuées (du type "livre rouge") présentent chacune un certain nombre d'inconvénients. Ceci conduit à rechercher un compromis entre ces solutions qui ne nous satisfont pas. Nous proposons pour cela de distinguer deux types de protection dans un système distribué : une *protection globale* et une *protection locale*. En effet, un système distribué est composé de sites qui doivent communiquer et donc partager des informations au travers de sujets et d'objets "d'intérêt commun" mais chaque site est aussi le lieu de multiples opérations qui ne dépendent que de l'état du site pris isolément. La protection des objets "d'intérêt commun"doit être différente de la protection d'objets locaux temporaires qui interviennent dans des opérations locales d'un site. La protection globale va donc viser à protéger des objets persistants, partagés entre les différents sites alors que la protection locale va plutôt viser à gérer et contrôler les droits d'accès des objets locaux temporaires.

Notre approche repose donc sur la coopération de plusieurs entités :

- La protection globale du système est assurée par un **serveur d'autorisation**. Ce serveur est conçu de la même façon que le serveur d'authentification et d'autorisation du projet Delta-4.
- La protection locale de chaque site est assurée par un **noyau de sécurité local**. Ce noyau de sécurité doit contrôler les accès à tous les objets locaux en fonction d'informations qui dépendent de la nature de l'objet. Si l'objet local est un objet persistant du système, le contrôle d'accès se fera en fonction d'informations qui proviennent du serveur d'autorisation. Si l'objet est un objet temporaire local, le contrôle d'accès se fera localement en fonction d'informations locales.

La collaboration du serveur d'autorisation et des noyaux de sécurité permet d'assurer une protection efficace sans existence de points durs dans le système : chaque noyau de sécurité ne fait confiance qu'au seul serveur d'autorisation qui est composé d'un quorum de sites dont une minorité peut défaillir sans conséquences sur le système.

Nous nous intéressons dans ce mémoire à la protection de systèmes construits à partir d'objets répartis. La notion d'objets que nous venons d'introduire dans le paragraphe précédent va nous amener à reconsidérer la vision classique

de la protection bâtie sur les traditionnelles notions de sujets et d'objets. Nous décrirons désormais un système d'objets distribués comme un ensemble d'objets, au sens des langages orientés objets. Ces objets possèdent des champs qui constituent leur état (et en ce sens sont des objets dans la vision classique de la protection) mais possèdent des méthodes et peuvent être clients d'autres objets dans le système en invoquant une de leurs méthodes (et en ce sens sont des sujets dans la vision classique de la protection). Nous obtenons dès lors une nouvelle matrice de droits d'accès du système possédant en principe les mêmes entrées pour les lignes et les colonnes.

Enfin, par rapport à l'architecture CORBA, nous montrerons dans ce mémoire que les mécanismes de protection que nous mettons en œuvre pour des systèmes d'objets distribués conduisent à un système qui reste compatible avec la vision de l'OMG.

Chapitre 2

Un schéma d'autorisation pour les systèmes d'objets répartis

Ce chapitre est consacré à la présentation d'un schéma d'autorisation pour les systèmes composés d'objets répartis. Nous présentons tout d'abord les raisons qui ont motivé l'élaboration de ce schéma d'autorisation et les objectifs que nous avons voulu atteindre. Après cette introduction, nous décrivons en détail ce schéma d'autorisation proprement dit et nous présentons ensuite un exemple illustratif. Enfin, nous donnons quelques éléments de réponse au problème de l'extension de ce schéma aux réseaux à grande échelle.

2.1 Objectifs

2.1.1 Distinguer deux niveaux de protection

Comme nous l'avons expliqué dans le premier chapitre, gérer la protection de tous les objets du système dans une matrice de droits d'accès centralisée est inimaginable en terme de performances. De même, gérer sur chaque site une matrice de droits d'accès aux objets locaux pose un sérieux problème de cohérence de la politique de sécurité. De plus, il nous semble que ces deux méthodes tendent à considérer de façon uniforme l'ensemble des objets qui

constituent un système réparti. Nous pensons, au contraire, que les objets qui s'exécutent dans ce système sont de nature différente et qu'il est intéressant d'en tenir compte et d'en tirer partie pour assurer la sécurité globale du système. Certains objets sont des objets persistants de grande granularité (tous les démons sous Unix par exemple) et certains sont créés dynamiquement pour exécuter une tâche précise et ponctuelle. Il nous semble que la protection doit fondamentalement séparer ces deux familles d'objets. Les objets persistants, qui sont des objets utiles au fonctionnement du système réparti tout entier doivent être gérés de façon centralisée. Les objets dynamiques temporaires, qui sont créés sur un site pour une action ponctuelle doivent être uniquement considérés localement. Ceci nous conduit à deux niveaux de protection : globale et locale, et au concept de **serveur d'autorisation** et de **noyau de sécurité**. Le serveur d'autorisation prend en charge la protection des objets persistants du système et un noyau de sécurité gère de façon autonome la protection d'objets locaux sur chaque site du système, tout en interceptant et contrôlant systématiquement tous les accès aux objets locaux. Nous détaillons dans la partie 2.2 ces deux entités.

2.1.2 Résoudre le problème de la confiance

Il nous faut à présent nous poser le problème de la confiance. On ne peut prétendre assurer la protection d'un système réparti sans se pencher précisément sur ce problème. En quelles entités plaçons-nous notre confiance et comment nous donnons-nous les moyens de la justifier? Dans le livre rouge par exemple [TNI87], la confiance est placée dans la base de confiance du réseau (NTCB). Outre le problème de cohérence que cela peut poser, cela signifie que chaque TCB du système est un point dur. Dans Kerberos [KN93], la confiance est placée dans le serveur d'authentification. Si ce serveur défaille, ou si un intrus en prend le contrôle, plus aucune sécurité ne peut être assurée dans le système. Aucun de ces cas ne nous semble satisfaisant puisque dans chacun, il existe un point dur.

Dans notre approche, nous attribuons une confiance limitée dans chaque noyau de sécurité. Conformément aux propriétés d'un noyau de sécurité que nous avons énoncées dans le premier chapitre, il doit être très difficile d'en prendre le contrôle. Néanmoins, dans notre approche, nous considérons que si un individu parvient à devenir maître d'un noyau de sécurité particulier, cette intrusion ne doit pas avoir de répercussion sur la sécurité globale du système. Ceci signifie que l'intrus est alors capable d'accéder comme il le désire aux

objets locaux, mais qu'il ne possède aucun moyen de fabriquer localement des privilèges pour accéder à des objets distants et qu'il ne peut pas non plus se faire passer pour un objet distant.

Il faut avoir une très grande confiance dans le serveur d'autorisation, puisque celui-ci est le gestionnaire de l'ensemble des droits d'accès à des objets persistants du système. Ainsi, un intrus qui se rendrait maître de ce serveur pourrait librement accéder à cet ensemble d'objets et mettre en péril la sécurité du système tout entier. Nous justifions cette confiance en utilisant la Fragmentation-Redondance-Dissémination : le serveur est composé de plusieurs sites administrés indépendamment, le contrôle d'un nombre limité de sites par un intrus ou des administrateurs malveillants ne met pas en danger la sécurité du système. Dans ce cas donc, le serveur d'autorisation n'apparaît pas comme un point dur du système.

2.1.3 Respecter le principe du moindre privilège

Il nous semble également primordial de pouvoir assurer qu'un objet exécute une action avec seulement les privilèges strictement nécessaires pour réaliser cette action. Cette notion signifie notamment que, lors du déroulement d'une opération complexe dans le système, nécessitant la collaboration de multiples objets, nous voulons assurer que chacun des objets effectue la partie de l'opération qui lui incombe en possédant uniquement les privilèges nécessaires à cette tâche. Accorder trop de droits à un objet peut permettre à cet objet d'exécuter des opérations qui peuvent constituer un danger pour la sécurité du système. De plus, nous voulons pouvoir gérer la propagation des droits au sein des objets de façon à respecter le principe du moindre privilège. Cette notion essentielle est en général très mal respectée dans les systèmes classiques. Nous donnons ici quelques exemples de ces systèmes en nous attachant à montrer en quoi ils ne sont pas satisfaisants.

2.1.3.1 Unix

Dans Unix, un processus actif dans le système s'exécute pour le compte d'un utilisateur et en ce sens, possède tous les droits de cet utilisateur. Chaque processus qui s'exécute possède ainsi un ensemble de droits dépassant largement les droits dont il a besoin pour réaliser l'ensemble de ses opérations. De plus, dans Unix, la notion de délégation de droits est très primaire et ne per-

met absolument pas de respecter le principe du moindre privilège. En effet, la délégation de droits est effectuée grâce à la notion de bit SUID : si le bit SUID est positionné sur un programme exécutable, le processus qui exécute ce programme obtient tous les droits du propriétaire du programme. C'est ainsi que fonctionne par exemple, le programme qui permet à un utilisateur de changer de mot de passe dans le système. Le fichier des mots de passe est uniquement modifiable par le super-utilisateur et le programme **passwd** est un programme dont le propriétaire est le super-utilisateur et dont le bit SUID est positionné. Ceci signifie que lorsqu'un utilisateur invoque ce programme, il obtient les privilèges du super-utilisateur pour l'exécution du programme et peut ainsi modifier le fichier des mots de passe. On peut donc constater qu'un utilisateur, pour obtenir le droit de modifier le fichier des mots de passe (un fichier unique), obtient tous les droits du super-utilisateur. Il est donc impossible de respecter le principe du moindre privilège avec une telle délégation de privilèges. Une des solutions proposées dans Unix est l'utilisation des groupes et la création de programmes possédant le bit SGID positionné (pendant l'exécution du programme, le groupe du processus l'exécutant devient le groupe du programme). Cette solution permet en effet d'éviter une délégation trop importante de droits en définissant précisément les groupes et les droits que chacun des groupes doit posséder. Le problème est que ce travail reste très fastidieux et donc très souvent peu ou mal utilisé.

2.1.3.2 Melampus

Un des objectifs du projet Melampus, mené dans les laboratoires d'IBM depuis 1990, est de réaliser un système à base d'objets capable d'effectuer des contrôles d'accès de fine granularité [LSC91, CLS92]. Le problème que veut résoudre ce projet est le suivant : dans un système multi-utilisateur, les utilisateurs n'ont a priori pas confiance les uns dans les autres. Ceci signifie en particulier que l'invocation d'une méthode d'un objet peut entraîner l'invocation de méthodes d'autres objets appartenant à d'autres utilisateurs en qui on n'a pas nécessairement confiance.

Les contrôles d'accès dans Melampus sont effectués à chaque invocation de méthode. De plus, est introduite la notion de *principal*; elle désigne toute entité dans le système à laquelle est accordée des droits. Ainsi, chaque invocation est réalisée au nom d'un principal appelé le *principal courant.* Chaque objet dans le système possède un propriétaire qui est un principal. Melampus définit des règles qui déterminent quand et comment est changé le principal

courant lors de l'exécution d'une opération dans le système: lors de l'invocation d'une méthode, le principal courant prend automatiquement la valeur du propriétaire de l'objet invoqué. À la terminaison de l'invocation, le principal reprend la valeur qu'il avait avant l'invocation.

Si les contrôles d'accès effectués dans ce système sont effectivement de fine granularité puisqu'ils sont réalisés à chaque appel de méthode, nous pouvons constater que ce schéma est en fait l'adaptation du principe du bit SUID d'Unix à un système composé d'objets répartis: à chaque invocation de méthode, le principal courant, qui est nécessairement un utilisateur, prend la valeur du propriétaire de l'objet invoqué. Ce modèle n'apporte donc pas d'amélioration concernant le principe du moindre privilège.

2.1.3.3 Les bases de données orientées objet

Nous nous appuyons ici sur le modèle proposé par Elisa Bertino dans [Ber92]. D'autres modèles d'autorisation auxquels nous apporterions sensiblement les mêmes critiques sont détaillés dans [ADG+92, RSC92].

Elisa Bertino, dans [Ber92], propose un modèle d'autorisation pour les bases de données orientées objets. Dans ce modèle, les droits accordés aux utilisateurs sont des droits d'invoquer des méthodes d'objets. Ce modèle a été défini en réponse aux faiblesses du modèle d'autorisation précédemment élaboré dans le système Orion. Dans ce système, l'autorisation d'invoquer une méthode d'un objet donne implicitement le droit d'accéder aux objets manipulés par cette méthode. Elisa Bertino précise qu'il faut profiter du mécanisme d'encapsulation du modèle objet et faire en sorte que tout accès se réalise sous la forme d'un appel de méthode. Toute autre manipulation doit être interdite. Bertino distingue les méthodes *publiques* et *privées*. Une méthode privée ne peut être invoquée que par certaines méthodes d'autres objets alors qu'une méthode publique peut a priori être invoquée par n'importe quel objet ou utilisateur. À chaque utilisateur est associée la liste des méthodes publiques qu'il est autorisé à invoquer. Un utilisateur u peut ainsi exécuter une méthode m seulement si m est publique et si u possède le droit d'exécuter cette méthode. Cette autorisation n'implique pas que u est autorisé à invoquer les méthodes que m invoque lors de son exécution. Soit m' une telle méthode. Si m' est publique, alors l'appel sera exécuté si u est autorisé à invoquer m'. Si m' est privée, l'appel sera exécuté si la méthode m fait partie des méthodes autorisées à invoquer m'. Ce modèle introduit également la notion de délégation de

droits et plus particulièrement la notion de *mode protégé* concernant l'autorisation d'exécuter une méthode. Si un utilisateur u accorde à un utilisateur u' le droit d'exécuter la méthode m en mode protégé, toutes les invocations de méthodes réalisées par m seront contrôlées en fonction de l'identité de u et non de u', lorsque u' exécute m.

Ce modèle a le souci d'éviter la distribution trop généreuse de droits lors de l'exécution d'une méthode et en ce sens, il essaie de respecter le principe du moindre privilège : un utilisateur invoquant une méthode m n'obtient que le droit autorisant cette invocation et doit explicitement demander l'autorisation d'invoquer les méthodes appelées par m. Néanmoins, de même que pour Melampus, cette vision de la protection nous paraît une vision trop "Unix". En effet, chaque exécution de méthode se fait pour le compte d'un utilisateur. Que l'exécution de fasse en mode protégé ou non (ce qui revient une nouvelle fois à utiliser le principe du bit SUID), elle n'est jamais effectuée pour le compte d'un objet mais bien d'un utilisateur.

2.1.3.4 Kerberos v5 et SESAME

Dans Kerberos v5 ou SESAME, qui supportent la notion de délégation de droits sous forme de *procurations* (*proxies* en anglais), il n'y a pas non plus respect du principe du moindre privilège. En effet, dans Kerberos v5, par exemple, un objet authentifié o peut déléguer des droits à un autre objet o' en lui permettant d'accéder à un serveur d'objets pour lui. Ceci signifie que lorsque l'objet o' va vouloir effectuer des opérations sur les objets du serveur, tous les contrôles d'accès vont s'effectuer en fonction de l'identité de l'objet o. Une procuration contient à la fois l'identité de l'objet qui accorde des droits et l'identité de l'objet à qui ces droits sont accordés ainsi que la liste de ces droits. En permettant de déléguer seulement certains droits précis, ces procurations ne possèdent pas l'inconvénient majeur du bit SUID d'Unix qui permet à un objet de se faire passer pour un autre. En ce sens, les procurations constituent une meilleure implémentation du principe du moindre privilège. Néanmoins, il est important de remarquer que pour créer une procuration, l'objet qui va déléguer des droits doit nécessairement posséder ces droits pour les accorder. Il nous semble que cette propriété est une restriction à l'implémentation du principe du moindre privilège. L'objet o doit pouvoir demander à o' d'exécuter pour lui une tâche qu'il ne pourrait pas exécuter lui-même directement. Par exemple, dans Kerberos, si un utilisateur u a le droit d'imprimer un fichier mais pas le droit de le lire, alors il ne peut déléguer le droit de lire le fichier

à un serveur d'impression. L'utilisateur *u* doit donc avoir le droit de lire un fichier pour l'imprimer, ce qui n'est pas nécessaire selon le principe du moindre privilège.

2.1.3.5 Conclusion

Le principe du bit SUID d'Unix est sans doute une des pires solutions à envisager quant au respect du principe du moindre privilège. Dans le modèle d'autorisation de Bertino ainsi que dans Kerberos v5 et SESAME, une amélioration est apportée dans le sens où il y a un contrôle du transfert des privilèges. Ce respect du principe d'atténuation des privilèges est abordé par Minsky dans [Min78]. Il y explique qu'assurer la protection d'un système consiste d'une part à accorder des droits à des entités et à être capable de vérifier ces droits, mais aussi à être capable de gérer la façon dont ces droits se propagent dynamiquement dans le système. Le principe de l'atténuation des privilèges précise que les privilèges dans un système ne doivent pas pouvoir augmenter lorsqu'ils sont transmis dans le système. Ceci nous semble être une première amélioration. Néanmoins, lorsque l'on envisage l'exécution d'une opération complexe dans un système réparti, il nous semble que la véritable application du principe du moindre privilège consiste à assurer que chaque objet doit exécuter sa partie de l'opération avec sa *propre identité* et qu'il doit l'exécuter avec le *seul privilège* autorisant cette partie de l'opération. Le modèle proposé par Elisa Bertino, de même que Kerberos ou SESAME, ne permettent pas ce type d'exécution : un objet s'exécute constamment pour le compte d'un utilisateur. Il nous semble que chaque objet du système doit avoir des droits qui lui sont propres en fonction de son rôle dans le système pour exécuter les opérations liées à cette fonction. Il doit pouvoir également recevoir des droits ponctuels pour exécuter une partie d'une opération complexe, et dans ce cas, le droit qui lui est transmis est un droit qu'il utilise en son propre nom et que l'objet qui le lui transmet ne possède pas nécessairement. Cette approche est impossible dans Kerberos ou SESAME où le mode de transmission de privilèges implique que l'objet qui transmet un droit à un autre objet a également nécessairement ce droit. En somme, il est important de donner des droits aux objets eux-mêmes (et donc de les considérer comme des "principaux" si l'on se réfère à la terminologie de Melampus) car cela permet une plus juste attribution des droits d'accès. En effet, lorsque l'on attribue seulement des droits aux utilisateurs et que toutes les opérations sont effectuées pour le compte d'un utilisateur, on est amené à leur attribuer des droits qui ne seront utilisés qu'à des fins de délégation. Il nous semble plus en

accord avec le principe du moindre privilège de les attribuer directement aux objets qui les utiliseront.

Le schéma d'autorisation que nous voulons élaborer doit ainsi mettre en œuvre ces principes. C'est pourquoi, nous allons définir, pour les opérations élémentaires mais aussi pour des opérations complexes des règles permettant de générer des droits pour les objets intervenant dans le déroulement de l'opération. Nous introduirons la notion de **droits symboliques** destinés à autoriser l'exécution d'opérations complexes mais aussi la notion de **coupons** qui constitue notre outil de délégation. Un coupon est un droit précis pour un objet, il peut être transmis d'un objet o à un objet o' sans que o possède le droit correspondant au coupon. De plus, lorsqu'un objet reçoit un tel coupon, il peut exécuter l'accès autorisé par ce coupon avec sa propre identité. L'ensemble de ces principes va nous permettre de mieux respecter le principe du moindre privilège.

Dans la suite de ce chapitre, nous expliquons tout d'abord le rôle de chacun des deux niveaux de protection que nous avons définis ainsi que leur mode de collaboration. Nous présentons ensuite le schéma d'autorisation à proprement parler. Nous donnons ensuite un exemple détaillé mettant en œuvre les principes de notre schéma.

2.2 Serveur d'autorisation et noyau de sécurité : deux niveaux de protection

2.2.1 Le serveur d'autorisation

Comme nous l'avons précédemment indiqué, le serveur d'autorisation doit assurer la protection globale du système, c'est-à-dire assurer la protection des objets persistants du système. La notion de serveur centralisé assurant un service global dans un système n'est pas une notion nouvelle : notre serveur d'autorisation peut se rapprocher du serveur de tickets (*Ticket Granting Server*) de Kerberos par exemple ou du serveur de privilèges de SESAME (*Privilege Attribute Service*) [Par91]. De même, le serveur d'autorisation peut tout à fait faire partie d'une architecture de type CORBA et collaborer avec le service de nommage (*Naming Service*) par exemple pour assurer la protection du système. Cependant, dans Kerberos, le serveur de tickets est essentiellement un serveur destiné à fournir un ticket à un objet lui permettant de garantir son

identité auprès des serveurs d'applications, l'essentiel de l'autorisation étant réalisée par les serveurs eux-mêmes. Nous pensons que laisser le soin à tous les serveurs d'applications de gérer l'autorisation de façon indépendante a toutes les chances de donner un système dans lequel la politique d'autorisation soit incohérente et donc un système difficile à utiliser. En fait, l'approche de Kerberos et de SESAME correspond à un schéma client-serveur mais n'est pas bien adaptée pour un système composé d'objets coopérant (qui peuvent être tour à tour, serveur ou client). Notre souci principal va donc être de résoudre ce problème de la protection d'objets distribués dans un système : le serveur d'autorisation a donc pour rôle de gérer l'ensemble des droits d'accès aux objets persistants du système et d'assurer une gestion cohérente de la protection de ce type d'objet.

Il est clair dès à présent que la fonction du serveur d'autorisation en fait un élément très sensible du système : la sécurité globale du système repose sur le bon fonctionnement de ce serveur. En particulier, il doit être fiable vis-à-vis des fautes accidentelles et vis-à-vis des fautes intentionnelles. Pour cela, nous appliquons à ce serveur, la Fragmentation-Redondance-Dissémination que nous avons présentée dans le chapitre précédent. Le serveur d'autorisation va donc être composé de plusieurs machines. Sur chacune d'elles, sont fragmentées et disséminées l'ensemble des informations confidentielles et sont simplement répliquées les informations non-confidentielles. Un mécanisme de mise en cohérence des traitements (par vote majoritaire) sera mis en œuvre à chaque fois qu'une opération devra être réalisée par ce serveur. L'ensemble de ces mécanismes nous permet de placer une confiance justifiée dans notre serveur d'autorisation.

Le serveur d'autorisation fonctionne de la façon suivante :

- Il reçoit toutes les requêtes destinées à un objet persistant du système.
- Il autorise ou non l'accès (nous détaillerons dans la suite de ce chapitre en fonction de quelles informations cette vérification est réalisée).
- Il fournit à l'objet émetteur de la requête un privilège lui permettant de prouver qu'il est bien autorisé à effectuer l'accès (cette preuve devra être présentée au noyau de sécurité situé sur le site de l'objet invoqué).

Comme le souligne ce dernier point, le serveur d'autorisation fonctionne en étroite collaboration avec l'ensemble des noyaux de sécurité. Nous présentons maintenant les différentes fonctionnalités de chaque noyau de sécurité.

2.2.2 Le noyau de sécurité

Chaque site du système doit posséder un noyau de sécurité. Chacun de ces noyaux de sécurité a un double rôle :

- Il est chargé d'intercepter tous les accès aux objets locaux et de vérifier que chaque requête est bien porteuse d'un privilège qui autorise l'accès.
- Il est chargé de gérer de façon autonome les droits d'accès aux objets temporaires locaux.

Il semble en effet totalement inapproprié de gérer les droits d'accès à des objets temporaires au niveau du serveur d'autorisation. Cette gestion serait particulièrement lourde et de plus, dégraderait de façon significative les performances. Mais comme nous voulons assurer une protection efficace de tous les objets s'exécutant dans le système, il est nécessaire de protéger les objets temporaires au même titre que les objets persistants. Cette gestion est donc réalisée par chaque noyau de sécurité.

La figure suivante représente l'architecture du système.

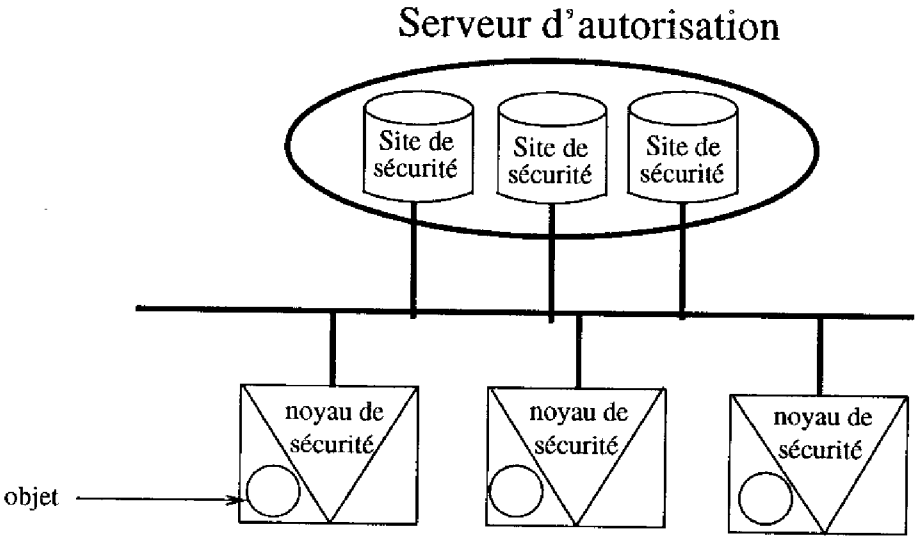


FIG. 2.1 – *Serveur d'autorisation et noyau de sécurité*

2.3 Le schéma d'autorisation

Dans cette partie, nous présentons le schéma d'autorisation que nous avons élaboré pour des systèmes composés d'objets répartis. Nous détaillons tout d'abord l'ensemble des principes et les propriétés essentielles de ce schéma puis nous expliquons à la fin de cette partie comment le serveur d'autorisation et les noyaux de sécurité interviennent et collaborent pour implémenter ce schéma.

2.3.1 Présentation des principales notions

2.3.1.1 Utilisateurs et objets (rôles et classes)

Les différentes entités que nous allons considérer dans notre système sont les utilisateurs et les objets. Les utilisateurs sont des personnes physiques pouvant se connecter au système et y effectuer des opérations. Le système est composé d'objets (au sens des langages orientés objets) qui peuvent donc être à la fois sujets et objets dans la définition classique de la protection. Nous éviterons ainsi dans la suite de ce mémoire d'utiliser le mot sujet, nous parlerons simplement d'objet, qui peut effectuer des opérations mais qui contient également un ensemble d'informations qui constituent son état. Pour réaliser la protection d'un système à objets distribués, nous utilisons ces notions d'utilisateurs et objets mais aussi les notions de classes et de **rôles**. La notion de classe nous permet de regrouper différents objets du système ayant les mêmes fonctionnalités (au sens de l'interface). Nous utilisons également les notions de sous-classe et d'héritage que nous avons définies dans le premier chapitre de ce mémoire. La notion de rôle permet de regrouper des utilisateurs ayant les mêmes privilèges. Un utilisateur peut avoir plusieurs rôles. Nous verrons dans la suite de cette partie, comment nous utiliserons ces différentes notions pour assurer la protection du système.

2.3.1.2 Activités et opérations de haut niveau

Nous définissons une **activité** comme une succession d'exécutions de méthodes de différents objets du système. Ces exécutions de méthodes sont fonctionnellement dépendantes et coopèrent dans le but de réaliser une **opération de haut niveau** du système, c'est-à-dire une opération qui requiert la coopération de plusieurs objets indépendants, par opposition à une **opération**

élémentaire qui est réalisée par l'exécution d'une méthode particulière d'un objet particulier. Dans la figure 2.2, un exemple d'activité est représenté. Cette activité réalise la tâche de haut niveau : enregistrement d'une scène. Cette activité consiste en :

1. Réaliser l'appel de la méthode *Enregistrer* de l'objet *Magnétoscope*,
2. Commencer l'exécution de la méthode *Enregistrer* de l'objet *Magnétoscope*,
3. Réaliser l'appel de la méthode *Filmer* de l'objet *Caméra*,
4. Exécuter la méthode *Filmer* de l'objet *Caméra*,
5. Retourner à la méthode *Enregistrer*,
6. Réaliser l'appel de la méthode *Sauvegarder* de l'objet *Cassette*,
7. Exécuter la méthode *Sauvegarder* de l'objet *Cassette*,
8. Retourner à la méthode *Enregistrer*,
9. Retourner à la méthode appelante de l'objet *Client*.

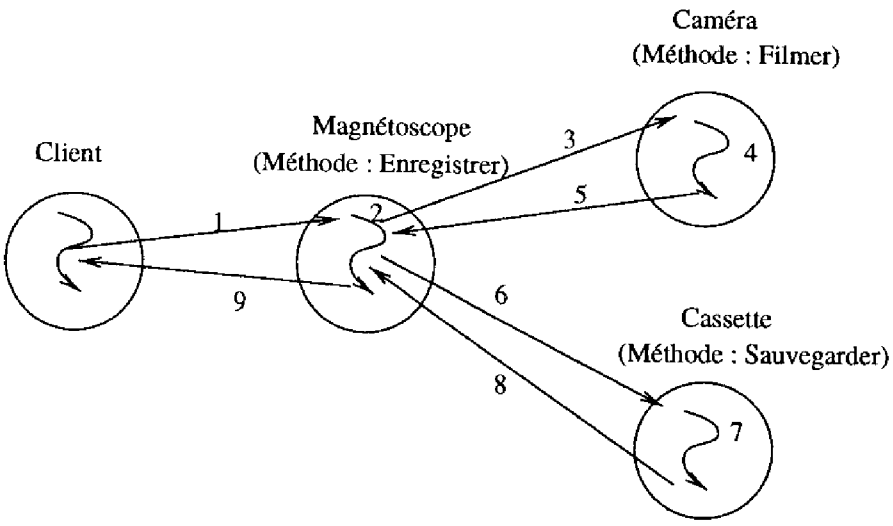


FIG. 2.2 – *Une activité*

Si nous examinons les différentes requêtes qui composent cette activité, nous pouvons établir les droits strictement nécessaires à la réalisation de cette dernière : l'objet *Client* doit obtenir le droit d'invoquer la méthode *Enregistrer* de l'objet *Magnétoscope* et l'objet *Magnétoscope* doit obtenir le droit d'invoquer la méthode *Filmer* de l'objet *Caméra*. La question que nous nous posons est la suivante : ces droits sont-ils des droits que les objets possèdent de façon persistante, qui leur ont été accordés à leur création, ou sont-ils des droits qui leur sont accordés pour réaliser cette opération particulière? Par exemple, un magnétoscope n'est pas nécessairement lié à une caméra pour enregistrer des données, il peut enregistrer à partir d'un téléviseur par exemple. Aussi ce droit d'accéder à la méthode *Filmer* n'est-il pas lié à l'opération de haut niveau qui est en train de se réaliser dans le système? Le but de cet exemple est donc de montrer qu'il nous semble nécessaire de pouvoir distinguer deux types de droits d'accès et que c'est grâce à cette distinction que nous parviendrons à respecter le principe du moindre privilège.

2.3.2 Les différents droits d'accès

2.3.2.1 Les droits de méthode et les droits symboliques

Le modèle de protection que nous présentons ici permet de distinguer les opérations élémentaires et les opérations de haut niveau et de leur faire correspondre des droits bien distincts. À une opération élémentaire est associé un **droit de méthode**. Ce droit correspond donc à l'exécution d'une méthode particulière d'un objet particulier. À une opération de haut niveau, est associé un ensemble de **droits symboliques**. Un droit symbolique représente un droit de haut niveau faisant intervenir plusieurs objets. Par exemple, le droit d'imprimer un fichier f sur toutes les imprimantes du système est un droit symbolique. En général, comme nous le montrerons dans 2.3.2.3, un ensemble de droits symboliques est nécessaire pour autoriser l'exécution d'une opération de haut niveau.

L'intérêt de distinguer opérations élémentaires et opérations de haut niveau est de rendre la gestion de la matrice des droits d'accès plus aisée pour les administrateurs de la sécurité. Ces derniers ont alors essentiellement à gérer des droits symboliques, et des règles définissant les privilèges qui seront accordés aux objets possédant ces droits. La suite de ce chapitre présente tout ceci en détail.

2.3.2.2 La matrice de droits d'accès

La matrice des droits d'accès que nous employons est une extension de la notion de matrice de droits d'accès "classique". Elle est analogue à la matrice de Lampson [Lam74] mais comporte à la fois des droits de méthodes et des droits symboliques. Chaque ligne de la matrice peut représenter un utilisateur, un objet, un rôle, ou une classe. Chaque colonne de la matrice peut représenter exactement les mêmes entités.

Les notions de rôle et de classe permettent de réduire les nombres de lignes et de colonnes de la matrice en regroupant les objets ayant les mêmes caractéristiques (appartenant donc à la même classe) et les utilisateurs ayant les mêmes privilèges (exerçant donc le même rôle dans le système). En fait, un rôle est un attribut de chaque utilisateur et un utilisateur peut avoir plusieurs rôles différents correspondant à différents privilèges.

Les droits qu'un utilisateur, un rôle, un objet ou une classe possède sur un autre utilisateur, un rôle, un objet ou une classe sont indiqués dans la case située à l'intersection de la ligne correspondante et de la colonne correspondante[3].

La matrice gère les deux familles de droits différents : droits de méthode et droits symboliques :

- Un droit de méthode est un droit permettant d'invoquer (et donc d'exécuter) une méthode particulière d'un objet particulier. Les droits de méthode qu'une entité e possède sur une entité e' sont exprimés par la liste des méthodes correspondantes de e'. Par exemple, si $M(e, e') = (f, g)$ alors e a le droit d'invoquer les méthodes f et g de e' et n'est pas autorisée à en invoquer d'autres. Notons que dans le cas de droits de méthode, e' ne peut être qu'un objet ou une classe.
- La représentation des droits symboliques est détaillée dans le paragraphe suivant ainsi que les règles de droits symboliques.

2.3.2.3 Les règles de droits symboliques

3. Posséder un droit sur une classe signifie posséder un droit sur toutes les instances de cette classe.

Définitions

M représente la matrice des droits d'accès.
U représente l'ensemble des utilisateurs du système.
R représente l'ensemble des rôles du système.
O représente l'ensemble des objets du système.
C représente l'ensemble des classes du système.
$Rôle(u)$ représente l'ensemble des rôles de l'utilisateur u.
$Classe(o)$ représente la classe de l'objet o.
$SousClasse(C)$ désigne l'ensemble des classes composé de la classe C et de ses descendantes, c'est-à-dire
$SousClasse(C) = \{C\} \cup \{Z, Z \text{ est une sous} - \text{classe de } C\}$.

Les droits symboliques sont des termes de la forme : $\mathtt{ds}(\mathtt{arg_1}, \mathtt{arg_2}, ..., \mathtt{arg_n})$ où ds représente une action dans le système et où $\mathtt{arg_1}, \mathtt{arg_2}, ..., \mathtt{arg_n}$ représentent des objets, des utilisateurs, des classes ou des rôles ($\forall i, \mathtt{arg_i} \in C \cup O \cup U \cup R$). Par exemple, SendMail(FILE, dupont) représente le droit d'envoyer un courrier à l'utilisateur *dupont*, ce courrier provenant de n'importe quel fichier du système. Dans la matrice d'accès, un droit symbolique doit être placé dans une colonne correspondant à un des ses arguments. Si le droit symbolique est placé dans la colonne correspondant à l'argument $\mathtt{arg_i}$, alors ce droit sera noté $\mathtt{ds}(\mathtt{arg_1}, ..., \mathtt{arg_{i-1}}, \mathtt{this}, \mathtt{arg_{i+1}}, ..., \mathtt{arg_n})$. Par exemple, le droit symbolique SendMail(FILE, dupont), s'il est placé dans la colonne correspondant à la classe FILE, sera noté SendMail(this, dupont). La ligne de la matrice où est placé un droit symbolique représente l'entité à laquelle ce droit est accordé. Notons que cette entité peut être un objet, un utilisateur mais aussi une classe ou un rôle.

Une opération de haut niveau dans le système est représentée de la façon suivante : $\mathbf{op}(\mathbf{arg_1}, ..., \mathbf{arg_n})$. $\mathbf{A(op)(e, arg_1}, ..., \mathbf{arg_n})$ représente l'autorisation d'exécuter cette opération de haut niveau pour l'entité e (e pouvant être un objet, un utilisateur, une classe ou un rôle).

Une règle de droits symboliques décrit l'ensemble des droits symboliques qui doivent être présents dans la matrice des droits d'accès pour autoriser l'exécution d'une opération de haut niveau pour une entité e. Chaque règle de droits symboliques est exprimée à l'aide du formalisme suivant :

$$\frac{\mathtt{precond}}{\mathbf{postcond}}$$

où precond représente un ensemble de droits symboliques et où **postcond**

représente l'autorisation de réaliser l'opération de haut niveau. Plus précisément, chaque règle $R(e, op, ds, arg_1, ..., arg_n)$ est ainsi définie :

$$RS(e, op, ds, arg_1, ..., arg_n) : \frac{\mathtt{ds_{e,arg_1}(arg_1, ..., arg_n)}, ..., \mathtt{ds_{e,arg_n}(arg_1, ..., arg_n)}}{\mathbf{A(op)(e, arg_1, ..., arg_n)}}$$

Cette règle signifie que l'opération **op** sera autorisée pour l'entité e avec les arguments $arg_1, ..., arg_n$ si un ensemble de droits symboliques s'appliquant chacun à un des arguments peut être trouvé dans la matrice. Plus précisément, tous les droits symboliques qui composent la précondition sont évalués selon la fonction d'évaluation $eval_M$ relative à la matrice d'accès M. Cette évaluation est réalisée de la façon suivante :

$eval_M(\mathtt{ds_{e,arg_i}(arg_1, ..., arg_n)}) = VRAI \Leftrightarrow$
$\exists\ \alpha_1, ..., \alpha_{i-1},\ \alpha_{i+1}, ..., \alpha_n \in C \cup U \cup O \cup R,$
$M(e, arg_i) = \mathtt{ds}(\alpha_1, ..., \alpha_{i-1}, \mathtt{this}, \alpha_{i+1}, ..., \alpha_n)$
et $\alpha_k \tilde{\supseteq} arg_k\ \forall k \in 1, ..., i-1, i+1, ..., n.$

La relation $\tilde{\supseteq}$ ($\tilde{\supseteq} \subseteq (C \cup U \cup O \cup R)^2$) est définie comme suit :

$$a \tilde{\supseteq} b \Leftrightarrow \begin{cases} a = b & si\ \ a, b \in O; \\ Classe(b) \in SousClasse(a) & si\ \ a \in C, b \in O; \\ a = b & si\ \ a, b \in U; \\ b \in Rôle(a) & si\ \ a \in U, b \in R \end{cases}$$

Un droit symbolique faisant intervenir un utilisateur ou un rôle comme paramètre est une notion relativement nouvelle. Intuitivement posséder un droit symbolique sur un utilisateur va impliquer à un moment ou à un autre de l'exécution de l'opération de haut niveau l'octroi de droits élémentaires (droits de méthode) sur des objets concernant l'utilisateur. Ainsi prenons l'exemple du droit symbolique `SendMail(FILE,dupont)`. Ce droit peut faire partie par exemple, de l'ensemble des droits permettant de réaliser l'opération de haut niveau "envoyer un courrier à l'utilisateur *dupont* depuis le fichier f". Cette opération fait intervenir la modification du fichier de spoule[4] contenant le courrier de *dupont*. Des droits concernant la modification de ce fichier vont donc être nécessaires à un certain moment. Le droit symbolique

4. cf. Glossaire des termes officiels de l'informatique, 2ème édition, AFNOR, 1989.

`SendMail(FILE,dupont)` et les règles de création de capacités vont permettre de générer de tels droits.

Un utilisateur ou un objet peut demander à réaliser une opération de haut niveau dans le système:

- Si un utilisateur u désire réaliser l'action de haut niveau $\mathbf{op(arg_1, ..., arg_n)}$, le système vérifie la règle $RS(u, op, ds, arg_1, ..., arg_n)$ mais aussi $RS(r, op, ds, arg_1, ..., arg_n), \forall r \in Rôle(u)$. Si l'évaluation de la précondition d'au moins une de ces règles est "VRAI", alors l'utilisateur est autorisé à effectuer l'opération correspondante.
- De même, si un objet o désire réaliser l'action de haut niveau $\mathbf{op(arg_1, ..., arg_n)}$, alors le système vérifie la règle $RS(o, op, ds, arg_1, ..., arg_n)$ mais aussi $RS(c, op, ds, arg_1, ..., arg_n)$, $\forall c,\ Classe(o) \in SousClasse(c)$.

Exemple

Prenons un exemple de règle de droits symboliques, *RS(e,op,enregistrerscène,mag,cam,cass)*:

$$\frac{\mathbf{ES_{e,mag}(mag, cam, cass)}, \mathbf{ES_{e,cam}(mag, cam, cass)}, \mathbf{ES_{e,cass}(mag, cam, cass)}}{\mathbf{A(enregistrerscène)(e, mag, cam, cass)}}$$

A(enregistrerscène)(e,rec,cam,cass) représente l'autorisation de réaliser l'opération de haut niveau "enregistrer une scène avec la caméra *cam* et le magnétoscope *mag* sur la cassette *cass*". Ce droit peut être accordé à une entité e seulement si les droits symboliques suivants se trouvent dans la matrice d'accès pour e:
- le droit d'enregistrer une scène avec le magnétoscope *mag*, pour un ensemble de caméras qui comprend *cam* et un ensemble de cassettes qui comprend *cass*;
- le droit d'enregistrer une scène avec la caméra *cam*, pour un ensemble de magnétoscopes qui comprend *mag* et un ensemble de cassettes qui comprend *cass*;
- le droit d'enregistrer une scène sur la cassette *cass*, pour un ensemble de magnétoscopes qui comprend *mag* et pour un ensemble de caméras qui comprend *cam*.

La matrice d'accès suivante contient un ensemble de tels droits symboliques (MAGNÉTOSCOPE et CAMERA sont des classes).

	mag	cam	cass	...
u	DS1	DS2	DS3	...
mag		Filmer	Sauvegarder	...
...	...	...	...	...

Avec :
DS1 = ES(this,CAMERA,CASSETTE),
DS2 = ES(MAGNÉTOSCOPE,this,CASSETTE) et
DS3 = ES(MAGNÉTOSCOPE,CAMERA,this).

Notons que la matrice d'accès suivante contient des droits qui autorisent également l'utilisateur u à effectuer cette opération de haut niveau (on suppose que $Staff \in Rôle(u)$).

	mag	cam	cass	...
Staff	DS1	DS2	DS3	...
mag		Filmer	Sauvegarder	...
...	...	...	...	...

2.3.3 Génération et distribution de capacités

2.3.3.1 Introduction

Lorsqu'un objet est autorisé à réaliser une opération de haut niveau ou une opération élémentaire dans le système, il doit recevoir une preuve de cette autorisation sous la forme d'une capacité. Cette capacité sera consultée par un noyau de sécurité qui devra pouvoir en vérifier la validité.

Comme nous l'avons introduit dans le premier chapitre, les droits d'accès peuvent être définis sous la forme de listes de contrôles d'accès (associées aux objets) ou sous la forme de listes de capacités (associés aux sujets). Dans notre approche, nous utilisons ces deux mécanismes. La matrice des droits d'accès est stockée sous la forme de liste de contrôle d'accès et des capacités sont délivrées aux objets s'ils sont autorisés à effectuer un accès. Une capacité accompagne ainsi chaque requête émise par un objet.

Il n'y a pas à proprement parler de définition universelle du contenu d'une capacité. Traditionnellement, une capacité doit contenir la référence d'un objet et une liste de droits [Fab74] (la possession de cette capacité donne ainsi les droits contenus dans la liste sur l'objet en question). C'est ce type de capacité qui est utilisée dans des systèmes comme Hydra [WCC+74] ou Amœba [TMvR86]. Il est clair qu'il ne doit pas être possible de contrefaire une capacité ni de la rejouer. C'est pourquoi les capacités sont généralement protégées par des mécanismes de chiffrement. Il existe un autre type de problème concernant les capacités : il ne doit pas être possible à un détenteur d'une capacité de la donner à quelqu'un d'autre. Une solution à ce problème a été proposée par Li Gong dans [Gon89]. Cette solution consiste à insérer dans une capacité l'identité de l'entité pour laquelle elle a été fabriquée. Nous suivons cette approche dans notre modèle. Les capacités que nous générons dans notre système contiennent toujours l'identité de l'objet pour lequel elle a été générée. Plus précisément, une capacité est constituée de :

- l'identité de l'objet auquel est octroyée la capacité ;
- l'identité de l'objet sur laquelle la capacité donne des droits ;
- la référence de la méthode à laquelle cette capacité donne accès ;
- un "nonce" [5] qui permet d'assurer que la capacité n'est pas rejouable ;
- éventuellement, les références des paramètres d'appel de la méthode.

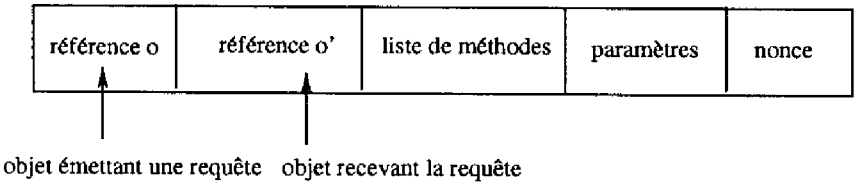


FIG. 2.3 – *Une capacité*

Justifions cette définition. La possession d'une capacité permet de prouver que l'on est autorisé à effectuer l'accès identifié dans la capacité. Or, dans notre schéma basé sur des objets, chaque accès se fait par l'intermédiaire de l'invocation d'une méthode. Une capacité contient donc la référence de l'objet

5. "Nonce", [NS78], signifie "used and only once" (utilisé une seule fois).

invoqué et d'une méthode (ou plusieurs), la méthode représentant l'opération autorisée sur l'objet.

Justifions, enfin, la présence de références de paramètres de la méthode. De façon générale, les paramètres d'une méthode peuvent être des types simples (entier, réel, caractère, etc.), mais aussi des références sur d'autres objets. Si l'on se réfère à l'exemple précédent, imaginons que l'utilisateur u soit autorisé à réaliser l'opération **enregistrerscène**($\mathbf{mag_1}, \mathbf{cam_2}, \mathbf{cass_3}$). Cette opération débute par l'invocation de la méthode *Enregistrer* de l'objet mag_1. Les deux paramètres de cette méthode sont la référence de la caméra et la référence de la cassette. Si la capacité fournie à l'utilisateur u ne comprenait pas la référence de cam_2 et de $cass_3$, l'utilisateur u pourrait invoquer cette méthode avec une autre caméra ou une autre cassette que celles pour lesquelles il a reçu l'autorisation de réaliser l'opération. Ces références sont par ailleurs utilisées par le magnétoscope pour accéder à la caméra et à la cassette. Donc si l'utilisateur u choisit judicieusement une référence de cassette pour laquelle l'opération de haut niveau **enregistrerscène** ne lui est pas accordée mais sur laquelle le magnétoscope mag_1 a des droits, il peut ainsi parvenir à écrire sur cette cassette. Il est donc nécessaire, lorsqu'une méthode admet comme paramètres des références d'objets, d'inclure dans cette capacité ces références. Dans l'exemple que nous examinons, la capacité contient les références de cam_2 et $cass_3$; le noyau de sécurité du site de mag_2 peut donc vérifier, au moment de l'invocation, que les paramètres de la méthode *Enregistrer* correspondent bien aux paramètres insérés dans la capacité.

2.3.3.2 La délégation de droits par coupons

La forme la plus courante de délégation de droits est la procuration. Une procuration est un privilège qu'un objet o peut transmettre à un objet o' pour que o' puisse réaliser une tâche pour le compte de o. Cette notion est relativement commune et est utilisée notamment dans Kerberos v5 [KN93] de même que dans SESAME [Par91] et CORBA [OMG95]. Pour l'exécution d'opérations de haut niveau, nous allons également avoir besoin de délégation de droits. Néanmoins comme nous l'avons expliqué dans l'introduction de ce chapitre, il nous semble que les procurations utilisées dans Kerberos ou SESAME ne permettent pas de strictement respecter le principe du moindre privilège puisqu'un objet ne peut déléguer qu'un droit que lui-même possède déjà et puisque l'objet qui reçoit une procuration exécute l'opération correspondante sous l'identité de l'objet qui lui a accordé cette procuration.

Pour mieux appliquer le principe du moindre privilège, nous définissons un nouvel outil de délégation, appelé **coupon**. Lorsqu'un objet o est autorisé à réaliser une opération de haut niveau, il reçoit une capacité lui permettant d'accéder à une méthode et il peut également recevoir des coupons. Ces coupons correspondent à des droits pour des objets qui réalisent une partie de l'opération de haut niveau. Ainsi o reçoit et transmet ces droits aux objets afin que ces derniers soient autorisés à effectuer leur partie de l'opération. Cette délégation de droits respecte au mieux le principe du moindre privilège puisque chacun des objets qui reçoit les coupons de o effectue sa partie de l'opération de haut niveau avec son identité et non celle de o. De plus, o n'a pas nécessairement les droits qu'il transmet aux autres objets (o peut transmettre le droit d'invoquer la méthode m d'un objet o'' à un objet o' alors que lui-même n'a pas ce droit). Par exemple, dans l'exemple de l'enregistrement d'une scène que nous avons présenté dans le paragraphe 2.3.1.2, l'objet *Client* autorisé à réaliser l'enregistrement de la scène, va recevoir pour l'objet *Magnétoscope*, le droit d'invoquer la méthode *Filmer* de l'objet *Caméra* alors que lui-même ne possède pas directement ce droit.

En fait, le terme *délégation de droits* est légèrement impropre dans notre schéma. Nous effectuons plutôt de la transmission de droits : un objet o reçoit un coupon créé pour o' et il le transmet à o' ; o ne fait que transmettre ce coupon à o', même si effectivement, grâce à ce coupon, on peut considérer que o' va pouvoir réaliser une opération dont o peut avoir besoin pour poursuivre son traitement.

Dans notre schéma, un coupon peut prendre deux formes. Il peut être soit une simple capacité, soit l'autorisation d'effectuer une opération de haut niveau, du type $\mathbf{A(op)(o', arg_1, ...arg_n)}$. Ceci signifie en fait que l'on peut aussi bien déléguer une opération élémentaire à un objet qu'une opération plus complexe.

2.3.3.3 Les règles de création de capacités

Les règles de création de capacités concernant des opérations élémentaires sont différentes des règles de création de capacités concernant des opérations de haut niveau.

Dans le premier cas, la règle de création de capacité correspondant à l'invocation de la méthode m de l'objet o' par l'objet o avec les paramètres $x_1, ..., x_n$ est représentée de la façon suivante :

$Cap(o, o'.m(x_1, ..., x_n)) ::= \{ref_o, ref_{o'.m}, nonce, ref_{x_1}, ..., ref_{x_n}\}$

Dans le second cas, la règle de création de capacités correspondant à l'opération de haut niveau $\mathbf{op(arg_1, ..., arg_n)}$ effectuée par l'objet o est représentée comme suit :

$Cap(\mathbf{o, op(arg_1, ..., arg_n)}) ::= \{liste\_de\_capacités, [liste\_de\_coupons]\}$

Nous pouvons tout d'abord remarquer que le nom "règle de création de capacités" est en fait légèrement impropre puisque des capacités mais aussi des coupons (qui ne sont pas nécessairement des capacités) sont délivrés par ces règles. Néanmoins, dans un souci de simplification de la terminologie que nous employons, nous avons choisi de conserver ce nom.

Il est important de constater que nous ne parlons ici que d'objets et non plus d'utilisateurs, de classes et de rôles. En fait, comme nous l'avons dit précédemment, un utilisateur ou un objet peut demander à effectuer une opération dans le système. Il peut être autorisé à effectuer cette opération en son nom propre mais aussi grâce à un de ses rôles (dans le cas d'un utilisateur) ou grâce à sa classe ou une de ses parentes (dans le cas d'un objet). Mais dès lors que cette autorisation est accordée, l'opération qui va en résulter dans le système est nécessairement réalisée par un objet (l'objet est la seule entité active dans notre système). Aussi, une capacité sera nécessairement délivrée pour un objet. Lorsqu'un utilisateur est connecté au système, il l'est nécessairement au travers d'un objet (comme un interpréteur de commandes par exemple). Aussi, si l'utilisateur est autorisé à effectuer une opération dans le système, la capacité correspondante sera délivrée à l'objet qui sert d'interface d'accès au système pour l'utilisateur.

2.3.4 Implémentation de ce schéma dans le système réparti

Les entités (serveur d'autorisation et noyau de sécurité) que nous avons présentées au début de ce chapitre, ont chacune un rôle précis dans la mise en œuvre de ce schéma.

2.3.4.1 Le rôle du serveur d'autorisation

La matrice des droits d'accès, les règles de droits symboliques et les règles de création de capacités sont gérées par le serveur d'autorisation. Celui-ci

reçoit tout d'abord toutes les requêtes qui concernent un objet persistant du système. Il analyse la requête qu'il reçoit et autorise ou non cette requête en fonction des informations qu'il possède (droits stockés dans la matrice des droits d'accès et règles). Seul le serveur d'autorisation possède l'ensemble de ces informations qui, comme nous l'avons expliqué, peuvent être fragmentées et disséminées sur l'ensemble des sites de sécurité qui le composent.

- Dans le cas d'une requête correspondant à une opération élémentaire, le serveur d'autorisation consulte simplement l'ensemble des droits de la matrice d'accès pour vérifier si le droit d'exécuter cette opération en fait partie ou non. S'il en fait partie, il génère grâce à la règle de création de capacités correspondante, une capacité. Il retourne cette capacité à l'objet qui a émis la requête.
- Dans le cas d'une requête de haut niveau, le serveur d'autorisation consulte la règle de droits symboliques correspondant à cette opération. Il détermine ainsi les droits symboliques qui doivent être présents dans la matrice pour autoriser cette opération. Il vérifie ensuite si ces droits existent ou non dans la matrice. S'ils sont bien présents, il fabrique les capacités et éventuellement les coupons correspondant à cette opération, ainsi que le lui dicte la règle de création de capacités correspondante. Il retourne cet ensemble de capacités et coupons à l'objet ayant émis la requête.

La figure 2.4 représente ce schéma de création de capacités.

2.3.4.2 Le rôle du noyau de sécurité

L'objet qui reçoit du SA (serveur d'autorisation) une capacité (et éventuellement des coupons) peut effectuer l'appel de méthode correspondant à cette capacité. Le noyau de sécurité situé sur le site de l'objet invoqué intercepte l'appel et vérifie que la capacité autorise l'accès. Si tel est le cas, l'invocation est effectivement réalisée.

Chaque noyau de sécurité gère également les droits d'accès aux objets temporaires locaux. Cette gestion est entièrement basée sur des capacités et est en fait totalement similaire à la gestion des capacités pour des objets persistants locaux. Chaque requête est interceptée et la capacité qui l'accompagne est vérifiée.

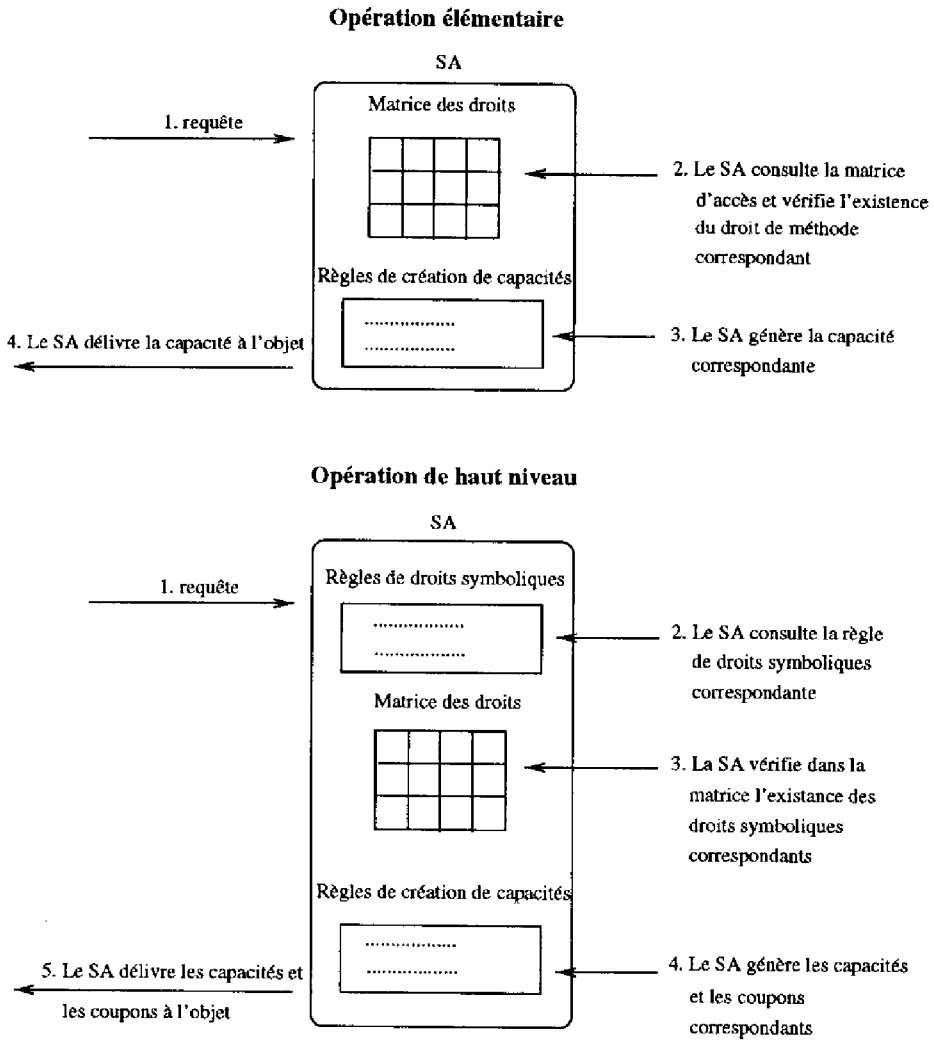


FIG. 2.4 – *Création de capacités par le serveur d'autorisation*

La figure 2.5 résume ce schéma de vérification de capacités.

Lorsqu'un objet demande la création d'un objet temporaire local, le noyau de sécurité local prend complètement en charge la gestion des droits de cet objet. Il crée l'objet demandé et retourne les capacités correspondant à toutes les

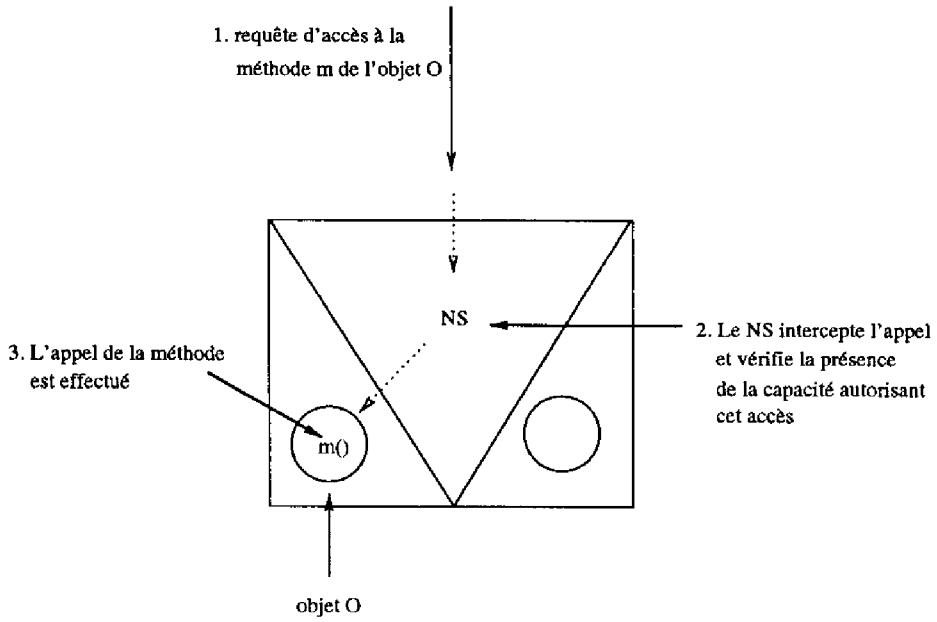


FIG. 2.5 – *Vérification des capacités par le noyau de sécurité*

méthodes de l'objet créé à l'objet créateur[6]. L'objet créateur peut alors, à son tour, demander au noyau de sécurité local de créer des capacités permettant à d'autres objets d'accéder à une méthode précise de l'objet créé.

Chaque noyau de sécurité est construit de telle façon qu'il soit très difficile à un intrus de s'en rendre maître. Ainsi, outre l'interception des différentes requêtes, chaque noyau de sécurité doit implémenter de solides protections au niveau de l'accès à toutes les ressources du site, c'est-à-dire des protections mémoires, des protections des disques et des divers périphériques. Ces protections sont des protections très classiques mais indispensables et peuvent en partie être réalisées par du matériel. Les protections sur les segments et les répertoires dans Multics [Kra80] en sont un bon exemple. Citons également les protections des segments mémoires offertes par les processeurs 386 ou 486 d'Intel. Enfin, dans le cadre du projet *Guide* mené à Grenoble, les travaux concernant l'adressage et la protection dans les systèmes répartis qui ont été effectués reposent principalement sur de solides protections mémoire [Hag93].

6. On peut également envisager de créer une capacité particulière donnant droit à accéder à toutes les méthodes d'un objet.

Revenons enfin sur le problème de la confiance que l'on accorde aux noyaux de sécurité, à présent que nous connaissons précisément son fonctionnement. Nous considérons qu'il est très difficile de se rendre maître d'un noyau de sécurité de par sa construction même, mais si un intrus parvient néanmoins à s'en rendre maître, la sécurité du système entier n'est pas compromise. En effet, si ce type d'attaque se produit, l'intrus peut autoriser ou refuser comme bon lui semble l'accès aux objets locaux en ignorant les capacités accompagnant les requêtes. De plus, il peut générer des capacités donnant des droits sur les objets temporaires locaux. Mais il ne peut en aucun cas générer des capacités permettant d'accéder à des objets distants. De telles capacités seraient refusées par les noyaux de sécurité distants qui eux, sont non corrompus. Il en est de même dans le cas d'une défaillance d'un noyau de sécurité due à une faute accidentelle.

2.4 Exemple

2.4.1 Introduction

Cette partie est consacrée à la présentation d'un exemple qui vient illustrer l'ensemble des principes que nous avons énoncés précédemment. Dans cet exemple, nous considérons qu'un utilisateur u veut imprimer un fichier f_3 sur une imprimante i_4. Les objets qui font partie de l'exécution de cette opération sont représentés dans la figure 2.6 : u est un utilisateur du système, si_1 est un serveur d'impression, instance de la classe *SERVIMP*, sf_2 est un serveur de fichiers, instance de la classe *SERVFICH*, f_3 et ft sont des fichiers, instances de la classe *FICHIER* et i_4 est une imprimante, instance de la classe *IMPRIMANTE*.

Nous présentons tout d'abord la matrice d'accès, les règles de droits symboliques et les règles de création de capacités relatives à cet exemple, puis nous décrivons le scénario du déroulement de cette opération.

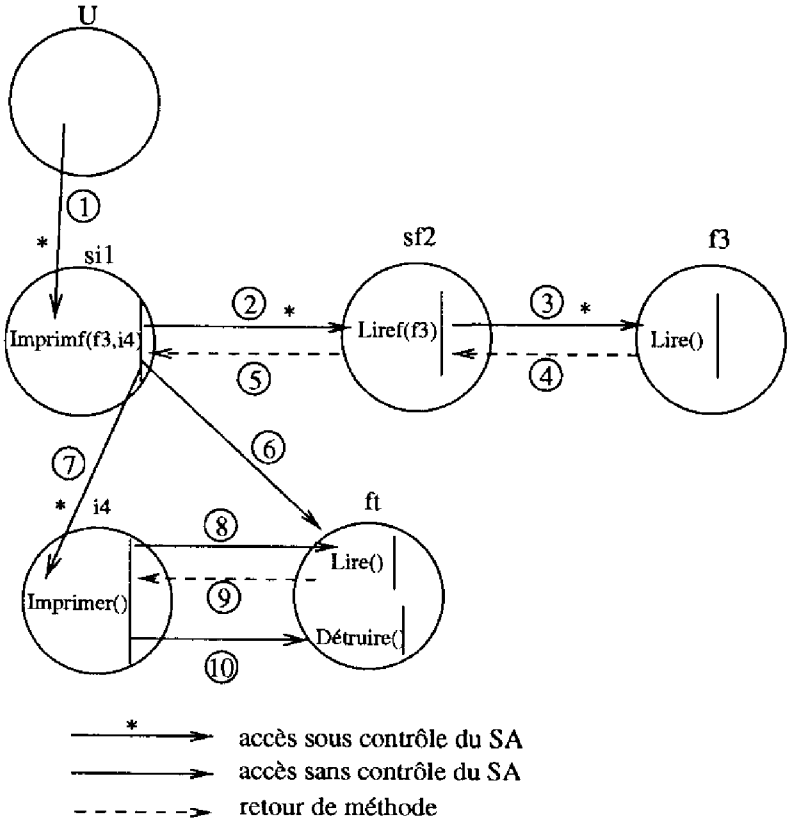


FIG. 2.6 – *Exemple d'opération de haut niveau*

2.4.2 La matrice de droits d'accès

	si_1	sf_2	f_3	i_4
u			`IF(this,IMPRIMANTE)`	`IF(FICHIER,this)`
si_1				Imprimer
sf_2			Lire, Écrire	

Définition des droits symboliques :

`IF(f,i)` : $(Classe(f) \in SousClasse(FICHIER)) \wedge$
$(Classe(i) \in SousClasse(IMPRIMANTE))$.

La première ligne de la matrice d'accès représente les droits symboliques qui sont accordés à l'utilisateur u. Le droit symbolique `IF(this,IMPRIMANTE)` dans la colonne f_3 signifie que u est autorisé à imprimer le fichier f_3 sur n'importe quelle imprimante du système (c'est-à-dire toute instance de la classe IMPRIMANTE). De la même façon, `IF(FICHIER,this)` dans la colonne i_4 signifie que u est autorisé à imprimer n'importe quel fichier sur l'imprimante i_4.

Les dernières lignes de la matrice d'accès représentent des droits de méthodes que si_1 et sf_2 ont respectivement sur i_4 et f_3 : si_1 est autorisé à invoquer la méthode *Imprimer* de i_4 et fs_2 est autorisé à invoquer les méthodes *Lire* et *Écrire* de f_3.

Le droit symbolique `IF` est défini pour s'appliquer à des classes particulières : le premier paramètre doit être une instance de la classe *FICHIER* ou une instance d'une classe descendante de la classe *FICHIER*. Le second paramètre doit être une instance de la classe *IMPRIMANTE* ou d'une de ses descendantes.

2.4.3 Les règles de droits symboliques

La règle suivante doit être incluse dans l'ensemble des règles de droits symboliques :

$$\text{(RS1)}\quad RS(e, imprimerfichier, IF, f, i) : \frac{\mathtt{IF_{e,f}(f,i)}, \mathtt{IF_{e,i}(f,i)}}{\mathbf{A(imprimerfichier)(e,f,i)}}$$

Cette règle signifie que pour être autorisé à exécuter l'opération "imprimer le fichier f sur l'imprimante i" (c'est-à-dire pour que la précondition de **imprimerfichier(e,f,i)** soit vraie), une entité e doit posséder dans la matrice d'accès les droits symboliques suivants :
- `IF` pour le fichier f et un ensemble d'imprimantes qui contient i;
- `IF` pour l'imprimante i et un ensemble de fichiers qui contient f.

Dans la suite de cet exemple, nous désignerons fréquemment une action de haut niveau par sa représentation. Nous parlerons par exemple, de l'action de haut niveau **imprimerfichier(f,i)** ou **lirefichier(f)**.

2.4.4 Les règles de création de capacités

L'ensemble de ces règles doit au moins contenir les règles suivantes :

(RC1) $\forall f$ *instance de* $FICHIER$, $\forall i$ *instance de* $IMPRIMANTE$,
$Cap(\mathbf{o}, \mathbf{imprimerfichier(f,i)}) ::=$
$\{Cap(o, \mathsf{ServeurImp}(i).Imprimerf(f,i)),$
$[\mathbf{A(librefichier)}(\mathsf{ServeurImp(i)}, \mathbf{f})]\}$

(RC2) $\forall f$ *instance de* $FICHIER$,
$Cap(\mathbf{o}, \mathbf{librefichier(f)}) ::= Cap(o, \mathsf{ServeurFich}(f).Liref(f))$

(RC3) $\forall si$ *instance de* $SERVIMP$,
$Cap(o, si.Imprimf(f,i)) ::= \{ref_o, ref_{si.Imprimf}, ref_f, ref_i, nonce\}$

(RC4) $\forall sf$ *instance de* $SERVFICH$,
$Cap(o, sf.Liref(f)) ::= \{ref_o, ref_{sf.Liref}, ref_f, nonce\}$

(RC5) $\forall f$ *instance de* $FICHIER$,
$Cap(o, f.Lire()) ::= \{ref_o, ref_{f.Lire}, nonce\}$

(RC6) $\forall i$ *instance de* $IMPRIMANTE$,
$Cap(o, i.Imprimer()) ::= \{ref_o, ref_{i.Imprimer}, nonce\}$

Dans chacune de ces règles, o représente un objet. À chaque méthode d'un objet est associée une capacité qui est donnée à chaque objet autorisé à invoquer cette méthode. Ainsi, $\{ref_o, ref_{i_4.Imprimer}, nonce\}$ est la capacité qui permet à l'objet o d'accéder à la méthode *Imprimer* de l'objet i_4. La notation ref_e représente une référence qui permet d'identifier l'entité e.

À chaque action de haut niveau est également associée une capacité permettant d'accéder à une méthode d'un objet. En effet, lorsqu'un objet a obtenu l'autorisation de réaliser une opération de haut niveau, il doit débuter cette opération par l'invocation d'une méthode particulière d'un objet précis. La capacité correspondante doit donc lui être fournie. La première partie de la règle RC1 indique que la capacité fournie pour débuter l'action **imprimerfichier(f,i)** est la capacité permettant d'accéder à la méthode *Imprimf* du serveur d'impression de l'imprimante i, tandis que la règle RC2 indique que la capacité fournie pour débuter l'action **librefichier(f)** est la capacité permettant d'accéder à la méthode *Liref* du serveur de fichier de f. Les notations ServeurImp(p) et ServeurFich(f) correspondent à des informations[7] indiquant

7. Ces informations sont conservées dans un serveur de répertoires qui peut être soit géré indépendamment, soit plus simplement inclus dans le serveur d'autorisation. Le fait d'associer le service de nommage et de répertoires au serveur d'autorisation réduit le nombre d'accès et justifie que le SA reçoive toutes les requêtes pour des objets persistants.

quel objet est le serveur d'impression de i et quel objet est le serveur de fichiers de f.

Comme nous l'avons vu précédemment, il est également possible d'associer à chaque opération de haut niveau un ou plusieurs coupons. Dans le cas de l'opération **imprimerfichier(f,i)**, le coupon [**A(lirefichier)**(ServeurImp(i), **f**)] représente l'autorisation, pour le serveur d'impression de l'imprimante i, de réaliser l'opération **lirefichier(f)**. Il est à ce propos important de noter que ce coupon correspond à un droit que u ne possède pas dans la matrice des droits d'accès (la règle de création de capacités correspondante ne vérifie d'ailleurs pas l'existence de ce droit). Ceci permet de respecter le principe du moindre privilège, puisqu'il n'est pas nécessaire d'être autorisé à lire un fichier pour l'imprimer. Ce coupon est donc délivré à l'objet autorisé à réaliser l'opération "imprimer le fichier f sur l'imprimante i". Cet objet passera ce coupon au serveur d'impression de i lorsqu'il l'accèdera à sa méthode $Imprim f$.

Dans notre exemple, la première requête effectuée par l'utilisateur u (ou plutôt par l'objet le représentant) correspond à demander l'autorisation d'effectuer l'opération "imprimer le fichier f_3 sur l'imprimante i_4". La première ligne de la matrice des droits d'accès ainsi que la règle des droits symboliques RS1 indiquent que cette opération est autorisée. Les règles de création de capacités RC1 et RC3 permettent de créer un message constitué d'une capacité et d'un coupon. Ce message est retourné à l'objet qui peut alors commencer l'opération.

2.4.5 Le scénario complet

Le scénario détaillé correspondant au déroulement de l'opération est décrit dans cette partie. Nous indiquons dans ce scénario toutes les manipulations de droits et nous précisons les rôles respectifs du serveur d'autorisation et des noyaux de sécurité. Les numéros de messages auxquels nous faisons référence sont représentés sur la figure 2.6.

- L'utilisateur u[8] demande au SA l'autorisation d'imprimer le fichier f_3 sur l'imprimante i_4. La ligne u de la matrice des droits d'accès contient les droits symboliques IF(f_3, IMPRIMANTE) et IF(FILE, i_4). La règle de droits symboliques RS1 autorise donc le déroulement de l'opération :

8. Ce que nous appelons ici "u"est en fait un objet de type "shell" s'exécutant pour le compte de u.

$$\frac{\mathbf{IF_{u,f_3}(f_3, IMPRIMANTE)}, \mathbf{IF_{u,i_4}(FICHIER, i_4)}}{\mathbf{A(imprimerfichier)(u, f_3, i_4)}}$$

- Selon la règle de création de capacités RC1, le serveur d'autorisation délivre à l'utilisateur u la capacité
$\{Cap(u, si_1.Imprimf(f_3, i_4)), [\mathbf{A(lirefichier)(si_1, f_3)}]\} =$
$\{ref_u, ref_{si_1.Imprimf}, ref_{f_3}, ref_{i_4}, nonce, [\mathbf{A(lirefichier)(si_1, f_3)}]\}$.

- u invoque la méthode $Imprimf$ de si_1 et présente la capacité $\{ref_u, ref_{si_1.Imprimf}, ref_{f_3}, ref_{i_4}, nonce, [\mathbf{A(lirefichier)(si_1, f_3)}]\}$ (**message 1**). Cette invocation est contrôlée par le noyau de sécurité situé sur le site de si_1 ; ce noyau vérifie que la capacité est valide. Si tel est le cas, l'invocation est réalisée et u transmet à si_1 le coupon $[\mathbf{A(lirefichier)(si_1, f_3)}]$.

- si_1 effectue alors l'opération $\mathbf{lirefichier(f_3)}$. Cette action est automatiquement interceptée par le serveur d'autorisation. Ce dernier consulte le coupon $[\mathbf{A(lirefichier)(si_1, f_3)}]$ et autorise donc l'accès. Notons ici également, que si_1 n'a pas besoin de posséder le droit de lire le fichier f_3 pour que cet accès soit autorisé. Les règles de création de capacités RC2 et RC4 précisent la capacité qui correspond à cet accès. Le serveur d'autorisation identifie donc sf_2 comme étant le serveur de fichier de f_3 et conformément aux règles RC2 et RC4 construit la capacité $Cap(si_1, sf_2.Liref(f_3)) = \{ref_{si_1}, ref_{sf_2.Liref}, ref_{f_3}, nonce\}$.

- si_1 invoque la méthode $Liref$ de sf_2 (**message 2**). Cette requête est accompagnée de la capacité $\{ref_{si_1}, ref_{sf_2.Liref}, ref_{f_3}, nonce\}$ qui est contrôlée par le noyau de sécurité du site de fs_2.

- fs_2 invoque la méthode $Lire$ du fichier f_3. Cette opération est interceptée par le serveur d'autorisation. Cette opération étant une opération élémentaire, ce dernier vérifie simplement que sf_2 possède dans la matrice des droits d'accès le droit de méthode correspondant. Ensuite, conformément à la règle RC5, il fabrique la capacité
$Cap(fs_2, f_3.Lire()) = \{ref_{fs_2}, ref_{f_3.Lire}, nonce\}$.

- Le noyau de sécurité situé sur le site de f_3 contrôle la requête émise par sf_2 (**message 3**). Il vérifie que la capacité qui l'accompagne ($\{ref_{fs_2}, ref_{f_3.Lire}, nonce\}$) autorise bien cet accès.

- si_1 récupère les données du fichier f_3 grâce à deux messages de retour (**messages 4 et 5**).

- si_1 crée un fichier temporaire local ft dans lequel il va écrire les données de f_3 (ce fichier est en fait utilisé comme un fichier de spoule). L'opération de création et d'écriture de ft est uniquement contrôlée par le noyau de sécurité local. Lors de la création du fichier, si_1 reçoit une capacité *propriétaire* qui lui permet d'invoquer toutes les méthodes de ft. Cette dernière lui permet ainsi de modifier le contenu de ft (**message 6**).

- si_1 invoque la méthode *Imprimer* de i_4.

- Le serveur d'autorisation intercepte cette requête et vérifie que si_1 possède dans la matrice des droits d'accès le droit de méthode correspondant. Comme tel est le cas, il délivre conformément à la règle RC5, la capacité $Cap(si_1, i_4.Imprimer()) = \{ref_{si_1}, ref_{i_4.Imprimer}, nonce\}$.

- Le noyau de sécurité situé sur le site de i_4 contrôle la requête émise par si_1 (**message 7**). Il vérifie que la capacité qui l'accompagne ($\{ref_{si_1}, ref_{i_4.Imprimer}, nonce\}$) autorise bien cet accès.

- si_1 demande alors au noyau de sécurité local de générer pour l'imprimante i_4 des capacités correspondant aux méthodes *Lire* et *Détruire* de ft. Cette demande est acceptée puisque si_1 peut présenter la capacité *propriétaire* sur l'objet ft.

- i_4 invoque la méthode *Lire* de ft (**message 8 et 9**). Cette invocation est contrôlée par le noyau de sécurité local qui vérifie que i_4 possède bien la capacité correspondant à cette méthode. Ensuite, i_4 imprime le fichier et le détruit (en invoquant la méthode *Détruire*, **message 10**, invocation autorisée grâce à la création de la capacité correspondante demandée par si_1).

2.5 Le stockage des informations : le serveur de répertoires

Après avoir défini le schéma d'autorisation, il est nécessaire d'étudier un modèle de représentation des données. En effet, il est important de pouvoir gérer de façon efficace l'ensemble des données que constituent la matrice des droits

d'accès et les différentes règles. Le modèle que nous proposons est basé sur la notion de **serveur de répertoires**.

Le serveur de répertoires permet de représenter les données sous forme d'un arbre dont les feuilles sont des descripteurs d'objets et les nœuds des répertoires. L'avantage d'un tel modèle est qu'il permet de hiérarchiser facilement l'ensemble des objets.

Chaque entrée du serveur possède un ensemble d'attributs nécessaires pour effectuer les contrôles. Par exemple, si l'entrée décrit un utilisateur, l'ensemble des attributs est au moins composé de :

- son nom ({Nom = Dupont}),
- son(ses) rôles ({Rôle = Professeur, Administrateur}),

Si l'entrée désigne un objet, les attributs incluent :

- son identité ({Id = laser1}),
- sa classe ({Classe = IMPRIMANTELASER}),
- ses méthodes accompagnées des règles de création de capacités correspondantes ({Méthodes = (Imprimer, RC6) ...}),
- une liste de contrôle d'accès
  ({ACL=(Tom,*imprimerfichier*(FILE,this)),
  (Administrateur,*imprimerfichier*(MANUEL,this))}),
- pour chaque droit symbolique s'appliquant à l'objet, une règle de droits symboliques et une règle de création de capacités.

D'autres attributs peuvent exister en fonction de la classe de l'objet. Par exemple, l'entrée d'une imprimante a un attribut qui désigne le nom de son serveur d'impression (ServeurImp="si1").

La matrice d'accès est en fait stockée par le serveur de répertoires sous forme de listes de contrôles d'accès (ACL) associées aux entrées des objets, des utilisateurs, des rôles et des classes dans le répertoire. Une ACL associée à un objet correspond en fait à une colonne de la matrice d'accès. Dans l'exemple ci-dessus, *Tom* peut imprimer n'importe quel fichier sur l'imprimante *laser1*

alors qu'un utilisateur ayant le rôle *Administrateur* peut imprimer seulement des manuels sur *laser1*. Pour chaque utilisateur ou rôle autorisé à effectuer une opération sur un objet, il existe une entrée correspondante (nom, droits d'accès) dans l'ACL de l'objet. Si le droit est un droit symbolique, des références sur la règle de droits symboliques correspondante ainsi que la règle de création de capacités correspondante apparaissent dans une entrée de chaque objet qui participe à la même action de haut niveau. Le serveur de répertoires associé à l'exemple de l'impression du fichier sur l'imprimante que nous avons détaillé est représenté dans la figure 2.7.

2.6 Extensions de ce schéma à des réseaux grande échelle

Le schéma d'autorisation que nous venons de présenter s'adresse plus particulièrement à des réseaux locaux. Le serveur d'autorisation correspond à une entité permettant d'assurer la protection d'objets dans un domaine. Néanmoins, nous pensons que ce schéma peut être facilement étendu à des réseaux grande échelle. Sans prétendre vouloir résoudre tous les problèmes, nous apportons dans cette partie quelques éléments de réponse.

L'extension de ce schéma à des réseaux à grande échelle passe nécessairement par la connexion de plusieurs serveurs d'autorisation. Pour de tels réseaux, composés de plusieurs milliers de machines, il est évidemment impossible de gérer tous les objets et utilisateurs à l'aide d'un seul serveur d'autorisation. Ainsi pour pouvoir gérer ce vaste ensemble, il est nécessaire de diviser les utilisateurs et les objets en domaines. Chaque domaine est alors géré par un serveur d'autorisation qui contrôle les accès à tous les objets de son domaine. Les utilisateurs d'un domaine sont authentifiés par le serveur d'authentification du domaine. Les informations relatives aux utilisateurs et aux objets d'un domaine sont stockées dans le serveur de répertoires correspondant. Un serveur de répertoires peut être facilement étendu et connecté aux autres serveurs de répertoires. X500 fournit un cadre pour réaliser cette connexion basée sur la notion d'alias [ITU93]. Un alias est un nom symbolique local qui désigne une entité qui n'est pas locale, c'est-à-dire dans notre cas, une entité située sur un autre serveur de répertoires. Quand on accède un objet dont la représentation locale dans le serveur de répertoires est un alias (donc un objet distant), la recherche est propagée sur les autres serveurs. Cela suppose donc l'existence, sur chaque serveur de répertoires, d'alias qui permettent d'accé-

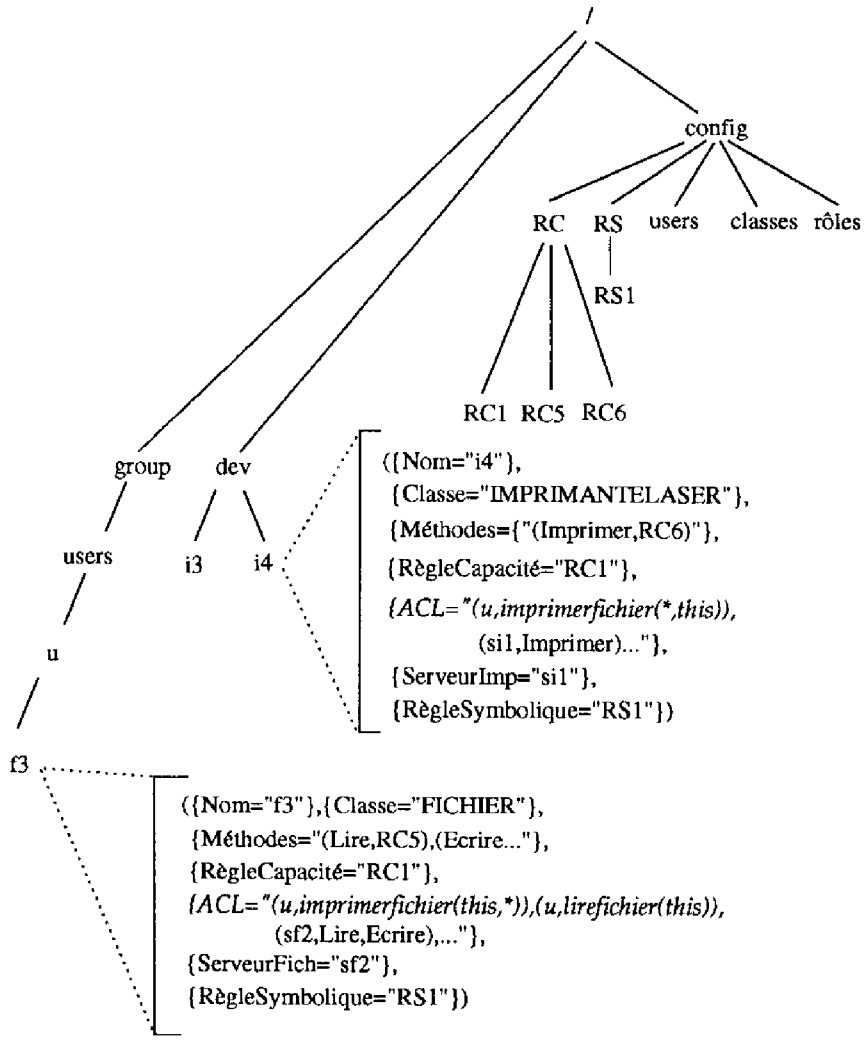


FIG. 2.7 – *Le serveur de répertoires*

der aux objets distants. Le schéma de nommage doit posséder les propriétés suivantes :

- la complétude : depuis n'importe quel serveur de répertoires, il doit être possible d'accéder à tout objet du système au travers d'alias ;

- la cohérence : des boucles infinies ne doivent pas exister ;
- la compréhensibilité : le schéma de nommage doit être aussi simple que possible et les noms des répertoires et alias prudemment choisis.

La figure 2.8 représente un alias d'un serveur de répertoires vers un autre serveur de répertoires.

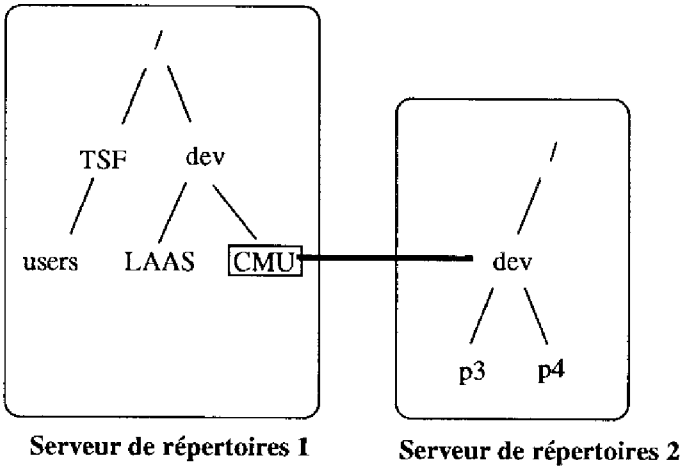


FIG. 2.8 – *Un exemple d'alias entre serveurs de répertoires*

Le principal problème que l'on rencontre lors de la connexion des serveurs de répertoires est le problème de la gestion de la cohérence. Un objet possède une seule entrée sur le serveur de répertoires local mais les autres serveurs de répertoires doivent être capables à tout moment d'obtenir des informations sur cet objet grâce aux alias. Ainsi, lorsqu'une mise à jour est effectuée, il est nécessaire de répercuter cette modification aux niveaux de tous les serveurs de répertoires concernés. Cette modification peut être réalisée de façon automatique en effectuant une diffusion de chaque modification.

Le protocole entre différents serveurs d'autorisation comporte les étapes suivantes :

1. le serveur d'autorisation local reçoit la requête d'un objet,
2. il consulte le serveur de répertoires local afin d'obtenir des informations sur les objets mis en jeu par la requête,

3. le serveur de répertoires récupère localement les attributs pour les objets locaux ; si un objet est géré par un autre serveur de répertoires, il l'interroge grâce à l'alias correspondant,

4. si le serveur de répertoires distant possède l'entrée demandée, il renvoie les attributs correspondants,

5. le serveur d'autorisation vérifie les droits de l'objet et, le cas échéant, fabrique une capacité pour l'objet ayant émis la requête.

Ainsi que nous pouvons le constater, la décision finale est prise par le serveur d'autorisation local. Les autres serveurs se comportent uniquement comme des serveurs de répertoires dans lesquels le serveur d'autorisation local puise des informations. Chaque serveur de répertoires est donc capable d'autoriser ou refuser l'exécution d'une opération même s'il ne connaît pas a priori l'ensemble des objets impliqués dans cette opération. Ainsi, dans l'exemple que nous avons présenté, nous pouvons imaginer que l'imprimante i_4 soit située dans un domaine différent du domaine du fichier f_3. Supposons que la requête d'impression ait été faite dans le domaine du fichier f_3. Le serveur d'autorisation local consulte l'entrée relative à f_3 dans son arborescence locale. Parmi les différents attributs, il y trouve la règle de droits symboliques relative à l'opération **imprimerfichier**($\mathbf{f_3}, \mathbf{i_4}$). Il sait alors que cette opération est autorisée si l'utilisateur u possède un droit symbolique pour f_3 et pour i_4 . Localement, le serveur d'autorisation vérifie qu'un tel droit existe pour f_3 en consultant l'attribut correspondant dans l'entrée du serveur de répertoires. Ensuite, il vérifie de la même façon que ce droit existe pour i_4. Mais en effectuant cette vérification, l'entrée relative à i_4 étant un alias vers une entrée d'un autre serveur de répertoires, il interroge ce serveur de répertoires.

La sécurité d'un réseau grande échelle repose donc sur la sécurité de l'ensemble des serveurs d'autorisation. Comme nous appliquons à chacun de ces serveurs la Fragmentation-Redondance-Dissémination, nous éliminons la présence de points durs dans un tel système et les serveurs d'autorisation (et de répertoires) des différents domaines peuvent se faire mutuellement confiance.

2.7 Conclusion

Dans ce chapitre, nous avons présenté un schéma d'autorisation pour les systèmes composés d'objets répartis. Nous avons introduit de nouveaux droits

appelés droits symboliques et la notion de coupons qui nous permettent de respecter scrupuleusement le principe du moindre privilège et d'obtenir une grande souplesse et un bon pouvoir d'expression pour la gestion des droits concernant les opérations de haut niveau dans le système. Nous avons ainsi montré que ces nouvelles notions nous permettent d'assurer que chaque objet prenant part à l'exécution d'une opération de haut niveau peut exécuter sa tâche pour son propre compte et sans obtenir plus de privilèges que nécessaire à la réalisation de cette tâche. Ce modèle prend de plus en compte les spécificités du modèle objet en termes d'encapsulation de données puisque tout accès à un objet ne peut être effectué que via un appel de méthode sur cet objet.

Les mécanismes de protection que nous avons détaillés dans ce chapitre sont conformes à l'architecture CORBA, puisqu'ils assurent effectivement une protection de tous les objets dans le système, ils offrent une notion de délégation de droits et ils permettent de protéger les communications entre objets.

Ce schéma d'autorisation n'est en aucun cas lié à une politique de sécurité particulière et peut donc être utilisé dans le cadre de politiques discrétionnaires (comme nous venons de l'illustrer dans ce chapitre) mais aussi dans le cadre de politiques obligatoires. Le chapitre suivant est donc consacré à la présentation d'une politique de sécurité multiniveau adaptée au modèle objet. Nous montrerons comment de tels contrôles peuvent s'intégrer dans notre schéma d'autorisation.

Chapitre 3

Une politique de protection multiniveau adaptée au modèle objet

Ce chapitre est consacré à la présentation d'un modèle de politique de sécurité multiniveau adapté au modèle objet. Après avoir rappelé la définition et l'intérêt d'une politique obligatoire en nous appuyant sur le modèle de Bell et LaPadula, nous présentons ensuite le modèle que nous avons élaboré et nous terminons ce chapitre en illustrant l'utilisation de ce modèle par un exemple.

3.1 Présentation des politiques obligatoires de confidentialité autour du modèle de Bell-LaPadula

Dans la première partie de ce chapitre, nous rappelons tout d'abord les objectifs des politiques discrétionnaires et obligatoires et nous présentons plus particulièrement ce qu'apporte en terme de confidentialité l'utilisation d'une politique obligatoire. Nous détaillons ensuite le modèle de Bell et LaPadula que nous avons introduit dans le premier chapitre. Nous présentons enfin ses principales faiblesses.

3.1.1 Politique discrétionnaire et politique obligatoire

Une politique d'autorisation discrétionnaire permet au responsable de l'information de manipuler librement ses droits comme bon lui semble, *à sa discrétion*. Le "responsable"de l'information est généralement le propriétaire de l'information (cas typique d'Unix). Dans le cas de politiques discrétionnaires, le contrôle d'accès est uniquement basé sur l'identité dans le système du sujet effectuant une opération, éventuellement sur son rôle ou sur son appartenance à un groupe d'utilisateurs.

À partir d'une telle politique, plusieurs variantes peuvent être définies [NCS87] :

- Dans le schéma *propriétaire*, c'est le créateur de l'objet qui détient le pouvoir sur l'objet et qui est donc son propriétaire. Il peut donner des droits d'accès à d'autres sujets mais il ne peut pas donner le droit de modifier les droits.
- Dans le schéma *laissez-faire*, le propriétaire de l'objet a la possibilité de permettre à d'autres sujets de modifier les droits de l'objet.
- Dans le schéma *centralisé*, le super-utilisateur et lui seul a la possibilité de modifier les droits d'accès.

Le principal problème des politiques discrétionnaires est qu'elles ne permettent pas d'éviter les fuites d'information et donc ne peuvent en général convenir pour des systèmes où la confidentialité est primordiale comme les systèmes militaires par exemple. Dans un système sécurisé par une politique discrétionnaire de type *propriétaire*, le propriétaire d'une information ne peut empêcher cette information de se répandre dans le système dans la mesure où il accorde un droit de lecture à un utilisateur particulier du système. En effet, accorder un droit de lecture d'une information à un utilisateur lui permet de réaliser une copie de cette information (copie qui lui appartient donc). En fait, dans un système sécurisé à l'aide d'une politique discrétionnaire, la collusion d'un ou plusieurs utilisateurs permet toujours de transférer une information d'un sujet à un autre. Cette collusion peut être cependant très difficile à établir dans certains cas. Ainsi, dans la politique multi-catégories définie dans [Bla92], les utilisateurs et les objets sont regroupés en catégories, chaque catégorie possédant un utilisateur particulier appelé *gestionnaire de catégorie* (un utilisateur peut appartenir à plusieurs catégories, un objet n'appartient qu'à une seule catégorie). Dans ce modèle, le créateur d'un objet ne peut modifier

les droits de cet objet et s'il désire changer un de ses objets de catégorie pour le rendre accessible à un utilisateur d'une autre catégorie, il faut l'intervention des gestionnaires des deux catégories.

L'autorisation basée sur une politique discrétionnaire possède une faiblesse évidente : elle ne possède aucune protection contre les chevaux de Troie. Un processus qui s'exécute dans le système s'exécute pour le compte d'un utilisateur et possède tous les droits de cet utilisateur. En particulier, il peut modifier les droits d'accès sur tous les objets dont le sujet est propriétaire.

En revanche, les politiques obligatoires permettent l'établissement de règles incontournables qui permettent d'assurer une protection efficace contre la fuite d'information.

3.1.2 Le modèle de Bell-LaPadula

Nous expliquons en détail ici le modèle de Bell et LaPadula que nous avions simplement introduit dans le premier chapitre ce ce mémoire.

Le modèle de Bell et LaPadula [BL74, BL75] est un modèle qui représente un système comme un ensemble d'états dont les transitions sont effectuées en fonction de règles. L'état d'un système est constitué d'un ensemble de triplets qui définissent les accès courants des sujets du système sur les objets du système. Dans cette vision, un sujet représente une entité active tandis qu'un objet représente un conteneur passif d'informations. L'ensemble des droits d'accès définis dans ce modèle sont : *execute*, *read*, *append*, *write* :

- *execute* signifie : accès sans observation ni modification;
- *read* signifie : observation sans modification;
- *append* signifie : modification sans observation;
- *write* signifie : observation et modification.

Différentes classifications (et habilitations) sont définies. Elles sont typiquement : NON-CLASSIFIÉ, CONFIDENTIEL, SECRET, TRÈS-SECRET. Elles sont totalement ordonnées selon la relation $<$. À chaque objet du système sont associés une de ces classifications et un **compartiment.** Un compartiment est un ensemble de catégories, une catégorie (par exemple "nucléaire") représentant le

domaine de l'information. Une information peut éventuellement entrer dans plusieurs catégories. Le **niveau de sécurité** d'un objet désigne l'ensemble composé de sa classification et son compartiment. Il se matérialise sous la forme d'une étiquette ou **label** associé à chaque objet du système. Les niveaux de sécurité sont partiellement ordonnés selon l'ordre suivant : le niveau de sécurité d'une information I défini par sa classification c et son compartiment C est dominé par le niveau de sécurité d'une autre information I' défini par sa classification c' et son compartiment C' si et seulement si $c \leq c'$ et $C \subseteq C'$. Réciproquement, on dira que le niveau de I' domine celui de I si et seulement si les mêmes conditions sont remplies. Nous notons cette relation de dominance : $\preceq$. De même, la relation de dominance stricte est notée : $\prec$. Le niveau de sécurité de I est strictement dominé par le niveau de sécurité de I' si et seulement si $c < c'$ et $C \subset C'$.

À chaque sujet du système sont associés deux labels. Le premier (statique) est le **niveau de sécurité maximal du sujet**, il est composé de **l'habilitation** du sujet et d'un compartiment. Le second label est le **niveau de sécurité courant**. Il est composé de la classification courante du sujet et d'un compartiment. Le niveau de sécurité maximal d'un sujet représente le niveau de sécurité maximal des informations qu'il va pouvoir consulter. Le niveau courant représente le plus haut niveau des informations que le sujet a consultées précédemment dans le système, ce niveau est donc flottant. Le niveau de sécurité maximal d'un sujet domine toujours son niveau de sécurité courant.

Ce modèle comprend deux règles de contrôle d'accès :

- **La propriété simple :** un état est sûr si et seulement si, pour chaque sujet dans le système qui observe un objet, le niveau de sécurité maximal du sujet domine le niveau de sécurité de l'objet.
- **La propriété étoile :** un état est sûr si et seulement si, pour tout sujet qui observe un objet O_1 et qui modifie un objet O_2, le niveau de sécurité de l'objet O_2 domine le niveau de sécurité de l'objet O_1.

  La définition de la propriété étoile peut être reformulée en termes de niveau de sécurité courant :

  un état est sûr si et seulement si, pour chaque sujet exécutant un accès m sur un objet:

  - le niveau de sécurité de l'objet domine le niveau courant de sécurité

du sujet si m est l'accès *append*;

- le niveau de sécurité de l'objet est égal au niveau courant de sécurité du sujet si m est l'accès *write*;
- le niveau de sécurité de l'objet est dominé par le niveau courant de sécurité du sujet si m est l'accès *read*.

La propriété simple a pour but d'empêcher les utilisateurs d'accéder à des informations pour lesquelles ils ne sont pas habilités et la propriété étoile vise à empêcher les flux d'information d'un niveau de sécurité donné vers un niveau de sécurité inférieur dans le système.

3.1.3 Limites du modèle de Bell-LaPadula

3.1.3.1 Un modèle trop restrictif

Une des principales limites de ce modèle réside dans l'aspect trop restrictif de la propriété étoile. Par exemple, dans un système qui implémente une telle politique de sécurité, considérons un utilisateur dont le niveau courant est SECRET qui lit un fichier CONFIDENTIEL et qui réalise une copie de ce fichier. La propriété étoile impose que la classification de la copie soit supérieure ou égale à SECRET. Or, de façon évidente, la classification de l'information contenue dans la copie est la même que la classification de l'information contenue dans le fichier original. La propriété étoile impose donc à l'utilisateur de créer un fichier avec une classification qui ne correspond pas à la réelle classification de son contenu.

Le problème de cette surclassification de l'information dans le système nécessite l'intervention régulière d'administrateurs qui doivent déclassifier l'information. En déclassifiant une information, ils violent les règles de la politique de sécurité, ce qui est possible pour des *trusted subjects*. Il est donc nécessaire dans un tel système de faire confiance à un certain nombre d'administrateurs qui vont être autorisés à violer la politique de sécurité tout simplement pour pouvoir disposer d'un système utilisable.

3.1.3.2 Le problème des canaux cachés

3.1.3.2.1 Définition des canaux cachés

Comme nous l'avons expliqué dans le premier chapitre, les modèles de sécurité basés sur la notion de matrice d'accès s'intéressent aux accès aux objets par des sujets. Comme ces modèles ne permettent pas de prendre en compte tous les flux d'information dans un système, les modèles de contrôle de flux et de non-interférence gèrent le problème du flux d'information d'objets vers d'autres objets. Pour réaliser ces flux, des canaux légitimes ou des **canaux cachés** peuvent être employés :

- un canal légitime, ou ouvert, est un canal de communication utilisé pour les transferts autorisés ;
- un canal caché est tout canal de communication pouvant être exploité par un processus de transfert d'information de telle manière qu'il viole la politique de sécurité [DoD85].

Dans le contexte d'une politique de sécurité multiniveau, l'utilisation d'un canal caché pour le transfert d'information nécessite un utilisateur de haut niveau ayant accès à des informations confidentielles et émettant ces données sur le canal, un utilisateur de bas niveau écoutant sur le canal. Un moyen de codage de l'information ou une synchronisation de l'émetteur et du récepteur sont nécessaires dans la plupart des cas. On distingue deux types de canaux cachés :

- un *canal de mémoire* (*storage channel* en anglais) est un canal impliquant l'écriture directe ou indirecte d'un élément de stockage par un processus et la lecture directe ou indirecte de ce même élément par un autre processus ; ce type de canal ne nécessite pas de synchronisation entre émetteur et récepteur ;
- un *canal temporel* (*timing channel* en anglais) est un canal dans lequel un processus module sa propre utilisation des ressources du système afin de modifier le temps de réponse observé par un second processus, lors de l'accès aux mêmes ressources ; la synchronisation entre émetteur et récepteur est indispensable pour un tel canal.

L'élimination des canaux cachés dans un système est une tâche particulièrement difficile puisque leur existence est directement liée à la présence de ressources partagées dans un système. Or, les performances d'un système reposent sur le partage des ressources dont il dispose. De plus, ainsi que le précise Loepere dans [Loe94], bon nombre des accès aux ressources partagées sont effectués dans des cas légitimes et rien ne les distingue a priori des accès effectués dans le but de transmettre de l'information illégalement. Dans le cas des canaux cachés de mémoire, le livre orange précise qu'il est possible de les identifier et des les éliminer totalement (ce qui est obligatoire pour un système A1). En revanche, il n'est pas possible d'éliminer totalement les canaux temporels puisqu'ils sont directement liés au fonctionnement même du système. Ainsi, dans le cas des canaux cachés temporels, l'objectif est de limiter le débit maximal de transmission de l'information à travers ce type de canaux à un niveau inutilisable. Ce débit, ou *bande passante*, se mesure en bits par seconde et la valeur maximale acceptable qu'on décide de lui accorder dépend beaucoup du système envisagé. Le livre orange indique néanmoins qu'une valeur supérieure à 100 bits/seconde serait inacceptable pour un système prétendu de sécurité. Les travaux décrits dans [Cal95] ont cependant permis de construire un système ne possédant aucun canal caché mais au prix d'empêcher la coopération entre utilisateurs de niveau différent et de perdre certaines fonctionnalités.

Les méthodes qui permettent d'identifier les canaux cachés dans un système sont principalement basées sur l'analyse de programme et de spécifications de ces programmes. Citons à titre d'exemple, l'analyse du code du noyau de Xenix effectué par Tsai et al. dans [TGC90]. En matière de réduction de la bande passante pour les canaux cachés temporels, citons les travaux de Hu dans [Hu91], qui est basé sur l'introduction de bruit dans toutes les horloges pouvant être partagées entre les sujets d'un système.

3.1.3.2.2 Les canaux cachés et le modèle de Bell-LaPadula

Le modèle de Bell et Lapadula permet l'utilisation de canaux cachés. Tel qu'il était formulé originellement les règles du modèle permettaient la création de canaux cachés de mémoire et de canaux cachés temporels. D'après Landwehr [Lan81], Walter et al. ont établi une version du modèle dans laquelle les règles ne permettent pas la création de canaux cachés de mémoire mais dans laquelle cependant les canaux cachés temporels existent. Par exemple, un processus p_1 peut moduler son taux de pagination en fonction de la sensibilité des

informations qu'il observe. Un autre processus p_2 (dont le niveau courant de sécurité est inférieur à celui de p_1) peut observer les variations du taux de pagination de p_1 et les "déchiffrer"pour construire un message que p_1 est en train de lui transmettre. Aucune règle dans le modèle de Bell-LaPadula ne permet d'empêcher un tel flux d'information.

3.1.3.3 Conclusion

En résumé, le modèle de Bell-LaPadula est à la fois trop restrictif et trop laxiste. Il est trop restrictif puisqu'il empêche l'exécution d'opérations qui ne provoquent pas de flux d'information. Aussi, les systèmes qui choisissent d'implémenter ce type de modèle acceptent une dégradation du service même rendu par le système et implémentent beaucoup de fonctions à l'aide "d'utilisateurs de confiance" *trusted subjects*. D'un autre côté, ce modèle est trop laxiste puisqu'il ne possède pas de règles permettant de se prévenir de l'utilisation des canaux cachés.

3.2 Extensions au modèle de Bell-LaPadula

3.2.1 Le modèle de John Woodward

Dans [Woo87], John Woodward explique que le problème de la surclassification de l'information provient du fait que, dans les implémentations traditionnelles de systèmes sûrs tels que Scomp [Fra83], les sujets et les objets héritent du niveau de sécurité de leur créateur. Si nous considérons l'exemple présenté dans le paragraphe précédent, lorsque l'utilisateur se connecte avec un niveau courant de sécurité SECRET, le système lui crée un interpréteur de commandes SECRET. Lorsque l'utilisateur demande à réaliser la commande "copy", l'interpréteur de commandes crée un processus SECRET pour exécuter la commande. Ce processus est SECRET parce qu'il est créé par un interpréteur de commandes qui lui-même est SECRET. Le processus qui réalise la copie crée un nouveau fichier dans lequel il copie le fichier original CONFIDENTIEL mais ce nouveau fichier est créé SECRET. John Woodward propose donc d'associer deux labels à chaque processus et donnée dans le système. Le premier label (qu'il appelle label de sensibilité) est un label flottant qui contient le niveau *réel* des données contenues dans un processus ou fichier. Lors de la création d'une information dans le système, cette information doit avoir le

label de sensibilité le plus bas puisqu'elle ne contient aucune donnée (création avec réinitialisation). De la même façon, lors de la création d'un processus, son label de sensibilité doit représenter le niveau des données contenues dans son espace d'adressage. De plus, si un processus exécute un appel système de telle façon que son espace d'adressage est totalement réinitialisé, alors le label de sensibilité du processus doit représenter la sensibilité de ce nouvel espace d'adressage même si ce nouveau label est inférieur au précédent. Le second label associé à chaque objet et sujet du système est un label de sécurité exclusivement utilisé par les règles de la politique obligatoire. Ce label est appelé niveau de contrôle d'accès obligatoire (*Mandatory Access Control Level* ou MACL en anglais). Pour des raisons de facilité, on référencera ces deux labels par "label de sensibilité"et "label de sécurité". Un exemple d'un tel étiquetage est présenté dans la figure 3.1.

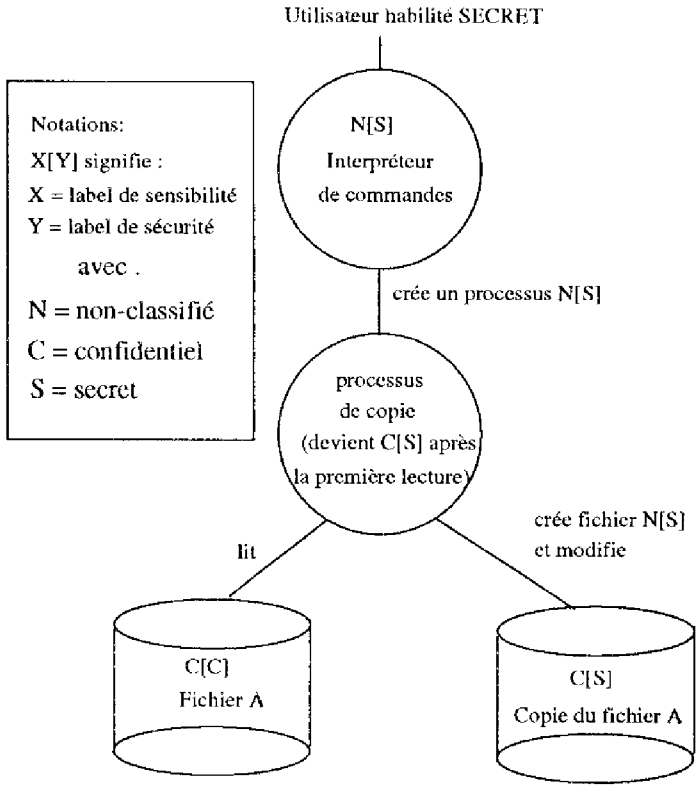


FIG. 3.1 – *Utilisation de deux labels*

John Woodward prétend que cette méthode permet d'étiqueter convenablement un fichier qui doit être exporté hors du système, ceci parce que le label de sensibilité représente le véritable niveau de sécurité de l'information contenue dans le fichier. Il explique également que l'intérêt de ce modèle est que chaque utilisateur dans le système peut repositionner (grâce à une commande privilégiée du système) le label de sécurité au niveau du label de sensibilité de façon à pouvoir partager ce fichier.

Ce modèle présente en effet quelques points intéressants dont notamment un avantage indéniable en ce qui concerne le processus de déclassification de l'information. En effet, ce processus ne nécessite pas, comme dans le modèle de Bell et LaPadula l'intervention "d'utilisateurs de confiance". Cette intervention est toujours gênante car ces "utilisateurs de confiance"doivent pour déclassifier l'information estimer quel nouveau niveau on peut lui attribuer, ce qui est une opération finalement très délicate et sans véritable garantie. Dans le modèle de Woodward, la déclassification peut s'effectuer de façon automatique et on peut avoir confiance dans le nouveau label de sécurité que l'on attribue à l'objet déclassifié puisqu'il correspond au niveau de sensibilité dont on est sûr de la validité.

Néanmoins, ce modèle possède l'inconvénient de créer des canaux cachés qui n'existent pas dans le modèle de Bell et LaPadula. En effet, considérons par exemple un utilisateur p possédant une habilitation SECRET qui désire transmettre de l'information à un utilisateur q dont l'habilitation est CONFIDENTIEL. L'utilisateur p déclassifie plusieurs fichiers de telle façon à faire passer leur label de sécurité de SECRET à CONFIDENTIEL (pour être autorisé à effectuer cette déclassification, l'utilisateur p a pris garde de ne pas écrire d'information d'un niveau de sensibilité supérieur à CONFIDENTIEL dans ces fichiers). Lorsqu'un fichier est déclassifié, l'utilisateur q peut le consulter alors que cela lui était impossible auparavant, et en particulier son nom par exemple. Il suffit que les deux utilisateurs aient convenu d'un code précis (par exemple, mot commençant par une voyelle signifie 1 et mot commençant par une consonne signifie 0) et un canal caché de mémoire est établi. De la même façon, des canaux cachés temporels peuvent être exploités lors du processus de déclassification.

Le modèle que nous allons présenter dans la suite de ce chapitre vise, de la même façon que le modèle de Woodward à être moins restrictif que le modèle de Bell et LaPadula mais il ne présente pas l'inconvénient d'introduire de nouveaux canaux cachés dans le système.

3.2.2 Le principe des dépendances causales

Bruno d'Ausbourg [d'A94] présente une façon originale de modéliser le flux d'information dans un système multiniveau. Ce modèle est basé sur la représentation d'un système comme un ensemble de points. Un point *(o,t)* représente un objet o à un instant t. Chaque point évolue avec le temps et chaque évolution est due à une transaction élémentaire effectuée dans le système. Une transaction peut ainsi modifier un point de telle façon qu'à l'instant t, l'objet o du point possède la nouvelle valeur v. Cet instant t et cette nouvelle valeur v dépendent fonctionnellement de points précédents. Cette dépendance est nommée *dépendance causale*. La dépendance causale du point *(o,t)* sur le point *(o',t')* avec $t' < t$ est représentée : $(o', t') \rightarrow (o, t)$. Le cône de dépendance est défini par :

$$\hat{cone}(o,t) = \{(o',t')/(o',t') \rightarrow^* (o,t)\}$$

où $\rightarrow^*$ désigne la fermeture transitive de la relation $\rightarrow$.

Les dépendances causales permettent de construire la structure des flux d'information à l'intérieur du système. Si un sujet connaît le fonctionnement interne du système, alors il est capable de connaître le schéma interne des dépendances causales. Donc, si un utilisateur connaît le fonctionnement interne du système et s'il peut observer un point de sortie x_0, alors il est capable d'en déduire des informations dans $\hat{cone}(x_0)$, c'est-à-dire, il est capable d'observer indirectement tous les points de $\hat{cone}(x_0)$.

Un système est considéré sûr si un sujet ne peut observer directement ou indirectement que les objets qu'il a le droit d'observer directement. Si nous notons Obs_s l'ensemble des objets qu'un utilisateur s peut observer directement ou indirectement et R_s l'ensemble des objets que cet utilisateur est autorisé à observer directement, nous pouvons dire que le système est sûr si : $Obs_s \subseteq R_s$.

Dans un système multiniveau implémentant une telle politique de sécurité, Bruno d'Ausbourg explique que deux conditions sont suffisantes pour garantir la sécurité du système. Dans un tel système, un niveau de classification *l(x)* est assigné à chaque point x et un niveau d'habilitation *l(s)* est assigné à chaque sujet s. La première condition stipule que l'habilitation d'un utilisateur s doit dominer la classification de l'ensemble des points Obs_s qu'il peut observer :

$$\forall s, x_0 \in Obs_s \Rightarrow l(x_0) \leq l(s)$$

La seconde condition stipule que si un objet y dépend causalement d'un autre

objet x, alors le niveau de classification de y doit dominer celui de x :

$$\forall x, \forall y, x \rightarrow y \Rightarrow l(x) \leq l(y)$$

Ce modèle est intéressant parce qu'il introduit une nouvelle façon de formaliser les flux d'information dans un système. Ce qui fait l'intérêt de cette formalisation est son aspect minimal : la notion de dépendance causale permet de décrire de la façon la plus stricte les flux d'information. Ainsi, si un système implémente une politique de sécurité basée sur cette notion, on est assuré que les règles du schéma d'autorisation décrivent les strictes conditions minimales qui doivent être vérifiées pour assurer qu'une opération s'effectue de façon sûre dans le système.

Le principal reproche que l'on peut faire à ce modèle est que les implémentations qui en ont été réalisées ne peuvent être utilisées que dans des cadres très spécifiques [Cal95], et que, de façon générale, il semble relativement difficile d'établir avec précision l'ensemble des dépendances causales dans un système. Cette recherche est de plus difficilement automatisable.

3.3 Modèles multiniveaux dans les bases de données orientées objet

Les travaux relatifs aux modèles de sécurité multiniveaux pour les bases de données à objets sont assez récents puisqu'ils ont réellement commencé à la fin des années 80. Nous allons tout d'abord donner les principales caractéristiques de ces modèles puis nous préciserons notre position par rapport à ces modèles.

On peut assez fidèlement résumer l'ensemble de ces travaux par l'opposition de deux courants : l'approche à *objets mononiveaux* et l'approche à *objets multiniveaux*.

3.3.1 L'approche à objets mononiveaux

L'approche à *objets mononiveaux* est principalement défendue par Jajodia et Kogan dans [JK90] et Millen et Lunt dans [ML92]. Il s'agit dans cette approche de considérer l'état de chaque objet comme l'information de base à protéger et donc à attribuer à chaque objet un unique niveau de sécurité. Cette approche implique donc qu'un utilisateur possédant un niveau de sécurité

supérieur à celui d'un objet peut lire l'ensemble des variables constituant l'état de l'objet. L'avantage de cette approche est qu'elle permet de s'adapter très facilement à la notion classique de noyau de sécurité qui implémente la protection de fichiers mononiveaux. L'utilisation de ce modèle ne nécessite donc pas l'implantation de mécanismes de protection supplémentaires à ceux qu'offre un noyau de sécurité standard. L'approche envisagé par Jajodia et Kogan est un peu particulière en ce sens où elle ne considère plus la notion de sujet mais seulement la notion d'objets. Jajodia et Kogan définissent un filtre de sécurité interceptant tous les messages dans le système (un message est la seule façon de transférer des informations dans le modèle objet) et autorisant ou refusant l'accès correspondant.

3.3.2 L'approche à objets multiniveaux

L'approche à *objets multiniveaux* est soutenue notamment par Lunt dans [Lun89], par Keefe, Tsai et Thuraisingham dans [KTT89] et par Boulahia-Cuppens, Cuppens, Gabillon et Yazdanian dans [BCCGY93a, BCCGY93b]. Dans cette approche, il s'agit de considérer chaque attribut d'un objet comme un récipient d'informations. Chaque attribut d'un objet est donc protégé de même qu'éventuellement l'objet lui-même. Lunt, dans [Lun89] présente un certain nombre de relations qui doivent exister entre le niveau de sécurité d'un objet, celui de sa classe et celui de chacun de ses attributs. Dans le modèle SODA développé par Keefe, Tsai et Thuraisingham un objet peut être mononiveau et dans ce cas un niveau de sécurité unique protège l'objet tout entier. Un objet peut également être multiniveau et dans ce cas, chaque attribut reçoit un niveau de sécurité protégeant l'accès à la valeur de l'attribut alors qu'un niveau de sécurité global protège l'existence de l'objet. De plus, deux niveaux de sécurité (un niveau courant et un niveau maximal) sont associés à chaque activation de méthode. Un certain nombre de règles autorisent ou refusent ces activations de méthodes en fonction des objets accédés.

L'approche que nous choisissons dans notre schéma d'autorisation est l'approche mononiveau. En effet, l'approche multiniveau nous semble être une approche plus intéressante dans le contexte de bases de données à objets que dans le contexte d'un système d'objets répartis qui est notre préoccupation. Cette vision où l'on désire protéger chaque attribut d'un objet séparément est finalement une adaptation d'une protection de chaque composant d'un tuple dans une base de donnés relationnelle. Cette vue correspond davantage à une approche client-serveur typique des bases de données dans laquelle un

client veut accéder à certaines données parmi une multitudes d'informations dont l'organisation doit être très finement structurée. Nous pensons que, dans un système d'objets répartis, chaque objet, tour à tour client et serveur, doit être considéré comme une entité à protéger. De plus, la protection d'objets multiniveaux dans un système réparti risquerait d'entraîner une complexité supplémentaire au niveau des contrôles et donc une dégradation de performances qu'on ne peut accepter dans ce type de système.

Notre modèle de sécurité rejoint en revanche ceux que nous venons de présenter au sens où nous pensons que les contrôles d'accès doivent s'effectuer au niveau de l'activation de méthodes. Ainsi, nous allons, un peu à la manière de SODA, attribuer aux différentes requêtes de notre schéma des niveaux de sécurité et utiliser ces niveaux de sécurité ainsi que ceux des objets pour établir les règles de notre schéma d'autorisation.

3.4 Une politique multiniveau adaptée au modèle objet

Le modèle que nous présentons ici a deux objectifs principaux : être adapté aux systèmes répartis modélisés sous forme d'objets et être moins restrictif que le modèle de Bell et LaPadula. Nous allons tout d'abord présenter les différentes entités qui composent notre modèle ; nous présenterons ensuite la façon dont nous étiquetons ces différentes entités pour pouvoir réaliser un contrôle multiniveau et nous finissons en présentant les règles de notre politique.

3.4.1 Définitions

Les principales entités qui entrent en jeu dans le définition de notre modèle sont : les activités, les utilisateurs et les objets. La notion d'activité a déjà été présentée au chapitre précédent. Nous ne revenons donc pas sur sa définition. Un utilisateur dans notre système représente une personne physique. Un objet, comme nous l'avons déjà expliqué représente toute entité de notre système susceptible de stocker de l'information mais aussi d'effectuer des traitements. Nous utilisons donc la notion d'objet au sens des langages orientés objet, composé d'un état et capable d'exécuter des appels de méthodes.

Une des caractéristiques principales de notre modèle est que nous identifions

deux sortes d'objets, qui nous semblent avoir des comportements différents en termes de stockage d'informations et que, par conséquent, nous allons pouvoir distinguer dans les règles de notre schéma d'autorisation. Nous appelons ces deux familles d'objets, **objets sans état** et **objets à état**. Dans l'exemple de l'activité présentée dans le chapitre précédent, l'activité véhicule de l'information depuis l'objet *Caméra* à l'objet *Cassette* (cf. figure 3.2). Cette information va bien sûr transiter par l'objet *Magnétoscope* mais ce dernier, après avoir transmis l'information à l'objet *Cassette*, n'en garde aucune mémoire. Son état est donc réinitialisé entre deux enregistrements successifs. En revanche, l'objet *Cassette* va conserver cette information dans son espace d'adressage. L'objet *Magnétoscope* est un objet sans état et l'objet *Cassette* est un objet à état.

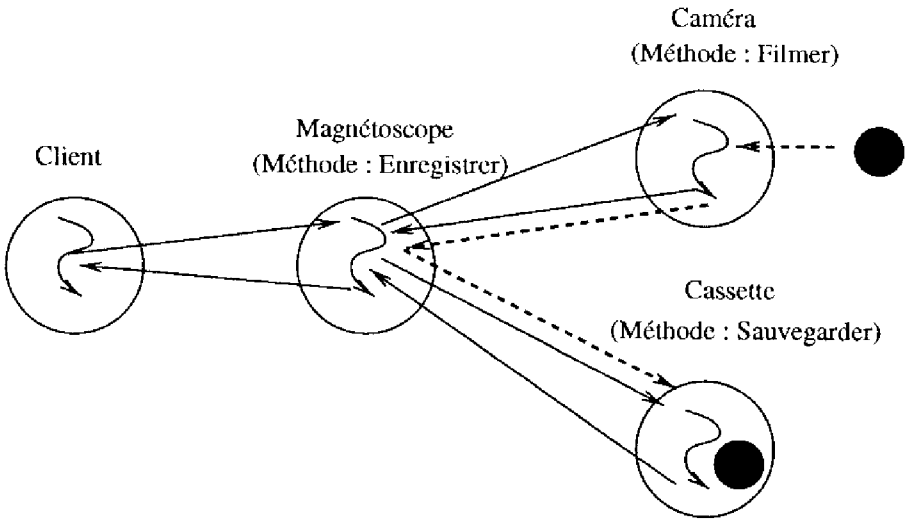


FIG. 3.2 - *Flux d'information dans une activité*

Nous pouvons maintenant généraliser cette notion : tout objet est capable de conserver des données dans son espace d'adressage mais ces données peuvent être classées en deux catégories : les données du système et les données ap-

plicatives. Les données applicatives sont les données qui composent l'état de l'objet du point de vue de l'application qu'il implémente, c'est-à-dire les données déclarées par le programmeur dans le code de l'application. Les données du système sont les informations qui sont ajoutées par le système d'exploitation à tout objet pour que celui-ci puisse s'exécuter dans le système (sur Unix par exemple, la pile d'un processus est un exemple de données du système). Un objet sans état est un objet qui ne garde aucune mémoire des données applicatives entre deux activations successives. Son état (au sens de l'application) est donc réinitialisé après chaque invocation dont il est l'objet. Cette propriété des objets sans état est très importante car cela signifie que deux requêtes successives qui accèdent le même objet sans état ne peuvent réaliser un flux d'information (ne peuvent s'échanger de l'information) grâce à cet objet. Cette notion d'objet sans état peut être rapprochée de la notion de réutilisation d'objet (*Object Reuse*) introduite dans le livre orange. À l'inverse, un objet à état est un objet qui stocke des données applicatives et dont les différentes méthodes consistent à consulter ou modifier ces données. Un fichier est l'exemple par excellence d'objet à état.

Si la notion d'objet à état semble naturelle et immédiate à tout programmeur (plus particulièrement des programmeurs des langages orientés objet où la notion d'état fait partie intégrante du concept de programmation), les objets sans état peuvent sembler déroutants. Cependant, un bon nombre d'exemples d'objets sans état peuvent être trouvés dans les systèmes actuels. Ainsi, le serveur NFS (Network File System) qui donne la possibilité sous Unix de créer un système de fichiers distribués de façon transparente est un objet sans état. Cet objet répond aux requêtes de montage de systèmes de fichiers sans en garder aucune mémoire par la suite. De même, tous les gestionnaires d'objets en général (les navigateurs par exemple) sont des objets sans état. Nous montrerons dans la suite de ce chapitre comment nous utilisons cette distinction entre objets à état et objets sans état pour obtenir un schéma d'autorisation moins restrictif que celui de Bell et LaPadula.

Le but d'une politique de confidentialité multiniveau est d'empêcher des utilisateurs d'obtenir des informations qu'ils ne sont pas habilités à observer. Comme nous supposons que chaque opération exécutée dans le système est réalisée au moyen d'une activité, nous devons empêcher les flux illégaux entre utilisateurs en empêchant les flux illégaux d'information entre les différentes activités du système. Nous devons donc assurer qu'une activité possédant un certain niveau de sécurité ne puisse transmettre de l'information à une autre activité dont le niveau de sécurité est inférieur (nous expliquons dans la

suite comment sont définis les niveaux de sécurité d'une activité). Or, deux activités peuvent échanger des données en accédant au même objet (l'une modifie l'objet et l'autre consulte l'objet après modification). Aussi devons-nous contrôler dans notre système toutes les interactions entre activités et objets, c'est-à-dire contrôler chaque requête qui accède à un objet. De façon à réaliser ces contrôles d'accès, nous devons attribuer des niveaux de sécurité aux différentes entités de notre modèle. Nous allons donc classiquement attribuer un niveau de sécurité à tout utilisateur du système. Comme les objets peuvent stocker de l'information, nous devons également leur attribuer un niveau de sécurité (nous attribuerons de façon différente des labels aux objets sans état et aux objets à état). De façon à contrôler chaque interaction entre les activités et les objets du système, nous devons enfin étiqueter les différentes requêtes.

3.4.2 Les différents labels

Nous définissons une ensemble fini de niveaux de sécurité partiellement ordonnés par la relation de dominance que nous noterons $\preceq$. Cette relation est la relation d'ordre partiel du modèle de Bell-LaPadula et la notion de niveau de sécurité sous-jacente est exactement la même que celle du modèle de Bell et LaPadula. Le niveau de sécurité comprend une classification ou une habilitation et un compartiment. Chaque niveau de sécurité se matérialise dans le système sous la forme d'un label, auquel nous ferons régulièrement référence. Par commodité, nous parlerons souvent seulement de la classification ou de l'habilitation lors de la définition des labels et de la présentation des règles, sachant qu'il faut y associer à chaque fois la notion de compartiment.

3.4.2.1 Les utilisateurs

À chaque utilisateur est assigné un unique niveau de sécurité L_u. Ce niveau représente son habilitation dans le système.

3.4.2.2 Les objets

Nous distinguons objets sans état et objets à état lorsque nous attribuons des niveaux de sécurité aux objets.

Un label unique L_O est associé à un objet à état O. Ce label représente le niveau de sécurité de l'objet, c'est-à-dire de l'ensemble des données qui composent l'état de l'objet. Ce niveau de sécurité est fixe et ne peut changer durant la durée de vie de l'objet.

Deux labels, $Lmin_O$ et $Lmax_O$, sont attribués à chaque objet sans état O, avec $Lmin_O \preceq Lmax_O$. $[Lmin_O, Lmax_O]$ représente un *intervalle de confiance* correspondant à la confiance que l'on estime pouvoir accorder à un tel objet[9]. Que représente cette confiance? Un objet sans état, comme nous l'avons expliqué précédemment, ne stocke aucune donnée applicative, et on ne peut donc lui assigner un label représentant la classification de son état. Cependant, un objet sans état peut être activé par une activité de la même manière qu'un objet à état. Chaque activité porte de l'information d'un certain niveau de sécurité. Chaque objet sans état peut donc potentiellement manipuler cette information. Le label $Lmax_O$ représente le niveau de sécurité maximal que peut lire l'objet sans état O. Dans le même esprit, le label $Lmin_O$ représente le niveau de sécurité minimal des données qu'un objet sans état peut écrire. Si on a une totale confiance en l'objet, on peut lui assigner les labels $[LMIN, LMAX]$, $LMAX$ étant le niveau de sécurité le plus élevé du système et $LMIN$ le plus bas. En revanche, supposons qu'un objet prétendu sans état puisse contenir un cheval de Troie qui conserve localement une copie de toutes les données portées par les requêtes qui l'accèdent. Le label $Lmin_O$ permet de garantir que la copie de ces données ainsi créée aura un niveau de sécurité supérieur ou égal à ce niveau. Ceci signifie qu'une autre activité devra nécessairement posséder un niveau de sécurité supérieur ou égal à ce niveau pour lire la donnée. Imaginons par exemple que l'administrateur d'un système décide d'utiliser un nouveau serveur NFS distribué librement et gratuitement sur un serveur de l'Internet. L'administrateur estime la confiance qu'il peut attribuer à ce serveur et ainsi lui associe un intervalle de confiance qui ne mette pas en danger les informations les plus sensibles de son système.

3.4.2.3 Activités et requêtes

Chaque activité dans le système véhicule de l'information et peut échanger de l'information avec les différents objets auxquels elle accède. Une activité doit donc porter un label. Nous associons une *parenthèse* de labels (c'est-à-dire deux labels) à chaque activité a. Ces labels sont notés $Lmin_a$ et $Lmax_a$. Ces labels sont des labels flottants et peuvent changer en fonction du mode

9. Un tel intervalle est ainsi défini : $\forall x \in [Lmin_O, Lmax_O] \Leftrightarrow Lmin_O \preceq x \preceq Lmax_O$.

d'accès et des labels des objets exécutés par l'activité. Le label $Lmin_a$ représente la classification de l'information véhiculée par l'activité, c'est-à-dire le niveau le plus haut des données qui ont été lues auparavant par l'activité. Le label $Lmax_a$ d'une activité représente son habilitation. Elle est initialisée avec l'habilitation de l'utilisateur qui démarre l'activité. Cette parenthèse de labels correspond donc à : $[classification, habilitation]$.

L'information véhiculée par une activité est en fait portée par les différentes requêtes (appels de méthodes) qui s'échangent entre les objets. Aussi, de façon à pouvoir réaliser des contrôles d'accès dans notre modèle, nous devons étiqueter ces requêtes. Nous attribuons une parenthèse de labels *fixes* à chaque requête r. Ces labels sont notés $Lmin_r$ et $Lmax_r$. Soit, par exemple, une activité composée de n requêtes $(r_1, r_2, ...r_n)$. La première requête r_1 accède à un objet pour exécuter une de ses méthodes. Cette exécution produit l'envoi d'un message, la requête r_2, dont les labels vont dépendre des labels de r_1, du type et des labels de l'objet traversés et des règles du schéma d'autorisation. Ainsi, les labels de toutes les requêtes composant l'activité vont être différents en fonction des objets traversés. La parenthèse de labels de l'activité $[Lmin_a, Lmax_a]$ (labels flottants) vaut successivement $[Lmin_{r_1}, Lmax_{r_1}]$,..., $[Lmin_{r_n}, Lmax_{r_n}]$.

3.4.3 Les règles du schéma d'autorisation

3.4.3.1 Modes d'accès en lecture et d'accès en écriture

Dans la présentation des règles de notre schéma d'autorisation, nous allons souvent faire référence aux modes d'accès en lecture ou d'accès en écriture. Ces expressions doivent être prises dans un sens bien précis. Nous appelons accès en lecture sur un objet à état O, toute exécution de méthode de O qui provoque un flux d'information de l'état de O vers l'activité qui est à l'origine de l'exécution de la méthode. Réciproquement, nous appelons accès en écriture toute exécution d'une méthode de O qui provoque un flux d'information de l'activité qui exécute la méthode vers l'état de l'objet O. Nous appellerons accès en lecture-écriture toute exécution de méthode qui provoque un échange d'information dans les deux sens entre l'activité qui exécute la méthode et l'état de l'objet O.

3.4.3.2 Principes du contrôle d'accès

3.4.3.2.1 Objets sans état

Une requête r peut accéder à un objet sans état O (en invoquant une méthode quelconque de O) si et seulement si l'intersection de $[Lmin_O, Lmax_O]$ et $[Lmin_r, Lmax_r]$ n'est pas vide.

Ceci signifie qu'un objet sans état dans lequel on a une confiance $[Lmin_O, Lmax_O]$ ne peut être invoqué par une requête que si cette dernière est habilitée à éventuellement recevoir des informations stockées par O ($Lmin_O \preceq Lmax_r$) ou si l'objet est habilité à éventuellement recevoir des informations de la requête ($Lmin_r \preceq Lmax_O$).

3.4.3.2.2 Objets à état

Les principes de notre politique de sécurité dérivent du modèle de Bell et LaPadula. Nous définissons donc de la même façon une propriété simple et une propriété étoile.

Propriété simple :

Une requête r de lecture sur un objet à état O est autorisée si et seulement si $L_O \preceq Lmax_r$.

En effet, le label supérieur de la requête représentant son habilitation, elle sera autorisée à lire un objet si son habilitation domine la classification de l'objet.

Propriété étoile :

Une requête r d'écriture sur un objet à état O est autorisée si et seulement si $Lmin_r \preceq L_O$.

En effet, puisque l'écriture est uniquement un flux d'information de la requête vers l'objet et que la classification de l'information portée par la requête est représentée par son label inférieur, il est nécessaire que ce label soit dominé par le label de l'objet pour que l'écriture s'effectue sans flux d'information illégaux.

Afin de pouvoir implanter ces règles, on attribue à chaque méthode d'un objet

à état un attribut qui indique quel mode d'accès est réalisé sur l'objet par l'exécution de la méthode: accès en lecture, accès en écriture ou accès en lecture-écriture.

3.4.3.3 L'évolution des labels

Dans cette partie, nous considérons la requête courante r d'une activité a qui accède un objet O.

- Si l'objet O est un objet sans état:
  - Si $Lmin_O \not\preceq Lmax_r$ ou si $Lmin_r \not\preceq Lmax_O$:
    l'accès est refusé. (R1)

    La relation $A \not\preceq B$ est équivalente à:
    $(B \prec A)$ ou (A *et* B *sont non comparables*). A et B sont non comparables si à la fois le compartiment de A n'est pas inclus dans celui de B et le compartiment de B n'est pas inclus dans celui de A.
  - Si $Lmin_O \preceq Lmax_r$ et $Lmin_r \preceq Lmax_O$:
    l'accès est autorisé avec restriction :
    $[Lmin_a, Lmax_a]$ devient
    $[max(Lmin_a, Lmin_O), min(Lmax_a, Lmax_O)]$ après l'accès. (R2)

    Cette restriction signifie que le niveau courant des informations portées par l'activité est remonté au niveau de confiance minimal accordé à l'objet et que l'habilitation de l'activité est abaissée au niveau de confiance maximal accordé à l'objet O.
    Notons également qu'un cas particulier de cette règle est le cas où l'intervalle $[Lmin_r, Lmax_r]$ est inclus dans l'intervalle $[Lmin_O, Lmax_O]$. Dans ce cas, l'accès est autorisé sans restriction, puisque :
    $max(Lmin_a, Lmin_O) = Lmin_a$ et
    $min(Lmax_a, Lmax_O) = Lmax_a$.

  La règle R2 décrit l'évolution de la parenthèse de l'activité lorsqu'elle accède l'objet O au moyen de la requête r. Si une activité accède un objet sans état dont le label inférieur domine la classification de l'information contenue dans la requête (représentée par le label inférieur de la requête) alors le label inférieur de la requête doit être élevé au label inférieur de

l'objet. Ceci représente le fait que le niveau courant des informations portées par l'activité doit être remonté au niveau minimal de l'objet sans état. Si une activité accède un objet sans état dont le label supérieur est dominé par l'habilitation de l'activité (représentée par le label supérieur de la requête) alors l'habilitation de l'activité doit être diminuée au label supérieur de l'objet pour empêcher des flux vers l'objet d'informations de niveau supérieur au niveau maximal autorisé pour l'objet. La figure 3.3 résume les règles de contrôle d'accès concernant les objets sans état.

- Si O est un objet à état :

  - Si la méthode m correspond à un accès en lecture et si $L_O \not\preceq Lmax_r$:
    l'accès est refusé. (R3)

    Cette règle traduit simplement le fait que l'activité n'est pas habilitée à lire l'objet.

  - Si la méthode m correspond à un accès en lecture et si $L_O \preceq Lmax_r$:
    l'accès est autorisé avec restriction, $Lmin_a$ devient $max(Lmin_a, L_O)$ après l'accès. (R4)

    Cette règle traduit le fait que le niveau courant des informations portées par l'activité est remonté au niveau de l'objet lu. Notons qu'un cas particulier de cette règle est le cas où $L_O \preceq Lmin_a$. Dans ce cas l'accès est autorisé sans restriction puisque :
    $max(Lmin_a, L_O) = Lmin_a$.

  - Si la méthode m correspond à un accès en écriture et si $Lmin_r \not\preceq L_O$, l'accès est refusé. (R5)

    Cette règle empêche un flux illégal d'information entre la requête et l'objet.

  - Si la méthode m correspond à un accès en écriture et si $Lmin_r \preceq L_O$, l'accès est autorisé sans restriction. (R6)

  - Si la méthode m correspond à un accès en lecture-écriture et si L_O ne se situe pas dans l'intervalle $[Lmin_r, Lmax_r]$, l'accès est refusé. (R7)

    Cette règle provient du fait que si $L_O \not\preceq Lmax_r$ alors la requête n'est pas habilitée à lire l'objet et si $Lmin_r \not\preceq L_O$, l'opération d'écriture conduit à un flux d'information illégal.

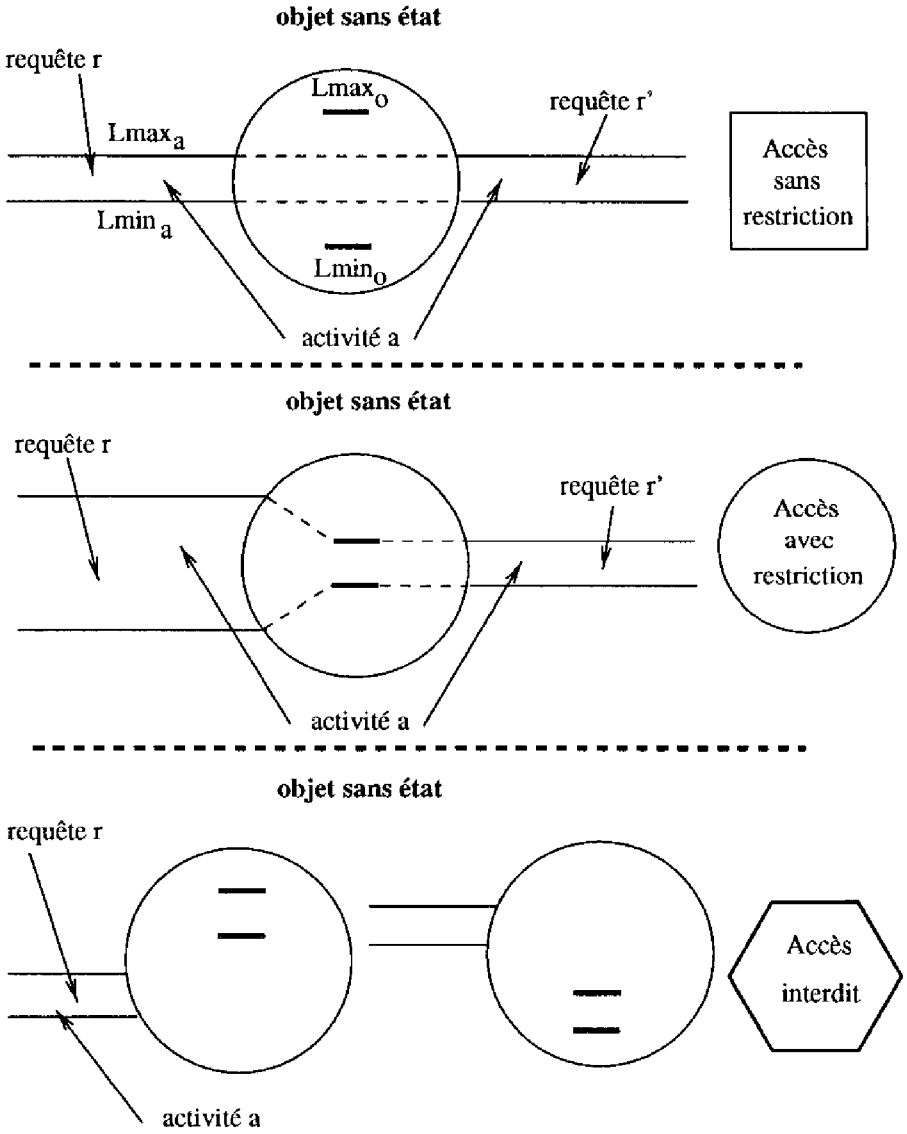


FIG. 3.3 – *Accès à un objet sans état*

– Si la méthode m correspond à un accès en lecture-écriture et si L_O se situe dans l'intervalle $[Lmin_r, Lmax_r]$:

l'accès est autorisé avec restriction, $Lmin_a$ devient $max(Lmin_a, L_O)$ après l'accès. (R8)

La restriction provient dans ce cas de l'opération de lecture, l'écriture étant autorisée sans restriction. Dans le cas où $L_O = Lmin_a$, l'accès est autorisé sans restriction.

Les règles R3 et R4 décrivent l'évolution de la parenthèse de l'activité qui effectue un accès en lecture sur un objet à état. Une activité ne peut pas lire un objet à état dont la classification domine l'habilitation de l'activité (R3). Si une activité effectue un accès en lecture sur un objet dont la classification domine la classification de l'information véhiculée par l'activité (représentée par le label inférieur de l'activité), alors le label inférieur de l'activité doit être élevé à la classification de l'objet (R4). Il n'y a pas de restriction à appliquer dans le cas d'un accès en écriture sur un objet à état, l'accès est soit autorisé sans restriction (dans le cas où la classification de l'objet domine le label inférieur de l'activité), soit refusé (dans le cas où la classification de l'objet est dominée par le label inférieur de l'activité). En effet, lors d'un accès en écriture, la classification de l'information portée par l'activité ne varie pas (nous avons précisément expliqué qu'un accès en écriture consiste uniquement à effectuer un flux d'information depuis la requête vers l'objet). Il n'y a par conséquent aucune raison de modifier la parenthèse de labels de l'activité qui est toujours porteuse de la même quantité d'information. La règle R6 présente le cas d'acceptation, sans restriction donc, de l'accès en écriture et la règle R5 présente le cas de refus de l'écriture (cette règle correspond à la propriété étoile du modèle de Bell et LaPadula).

Les figures 3.4, 3.5 et 3.6 résument l'ensemble des règles que nous venons de présenter à propos des objets à état.

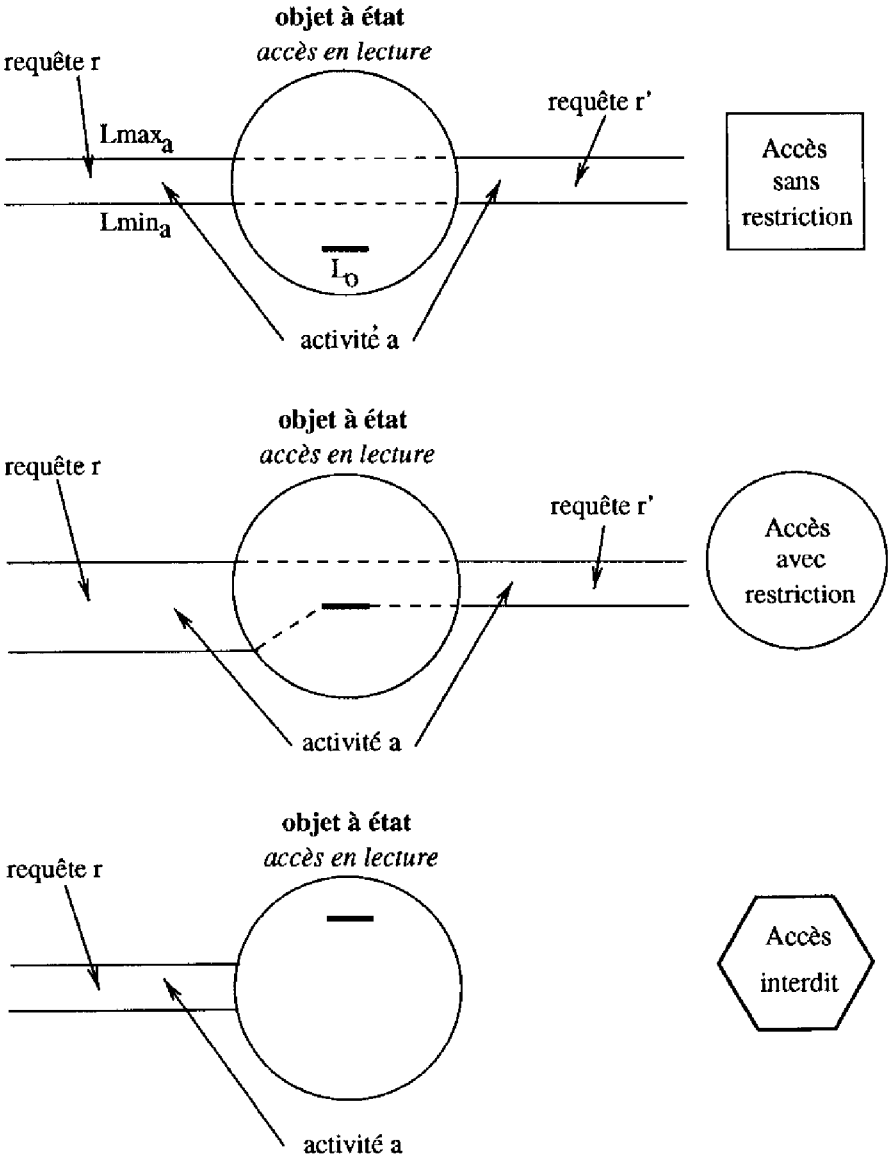


FIG. 3.4 – *Accès en lecture à un objet à état*

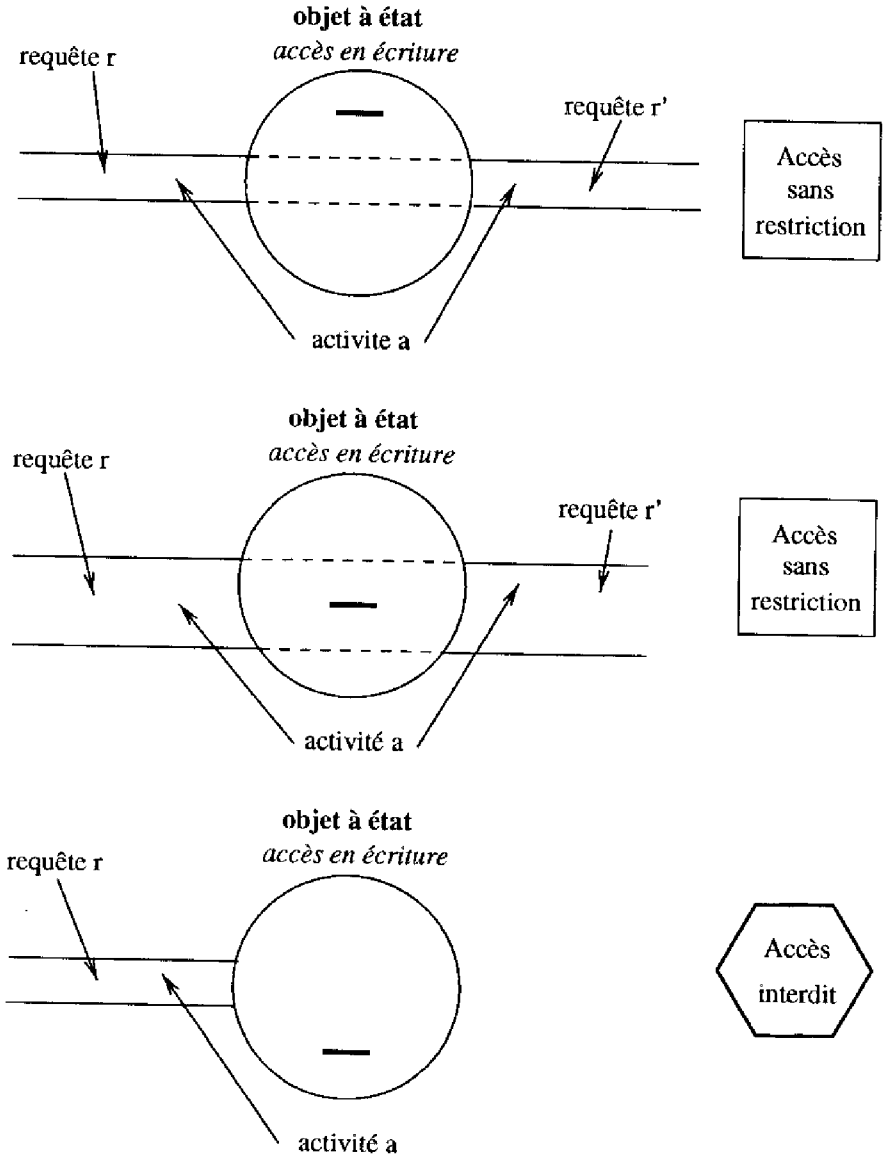


FIG. 3.5 – *Accès en écriture à un objet à état*

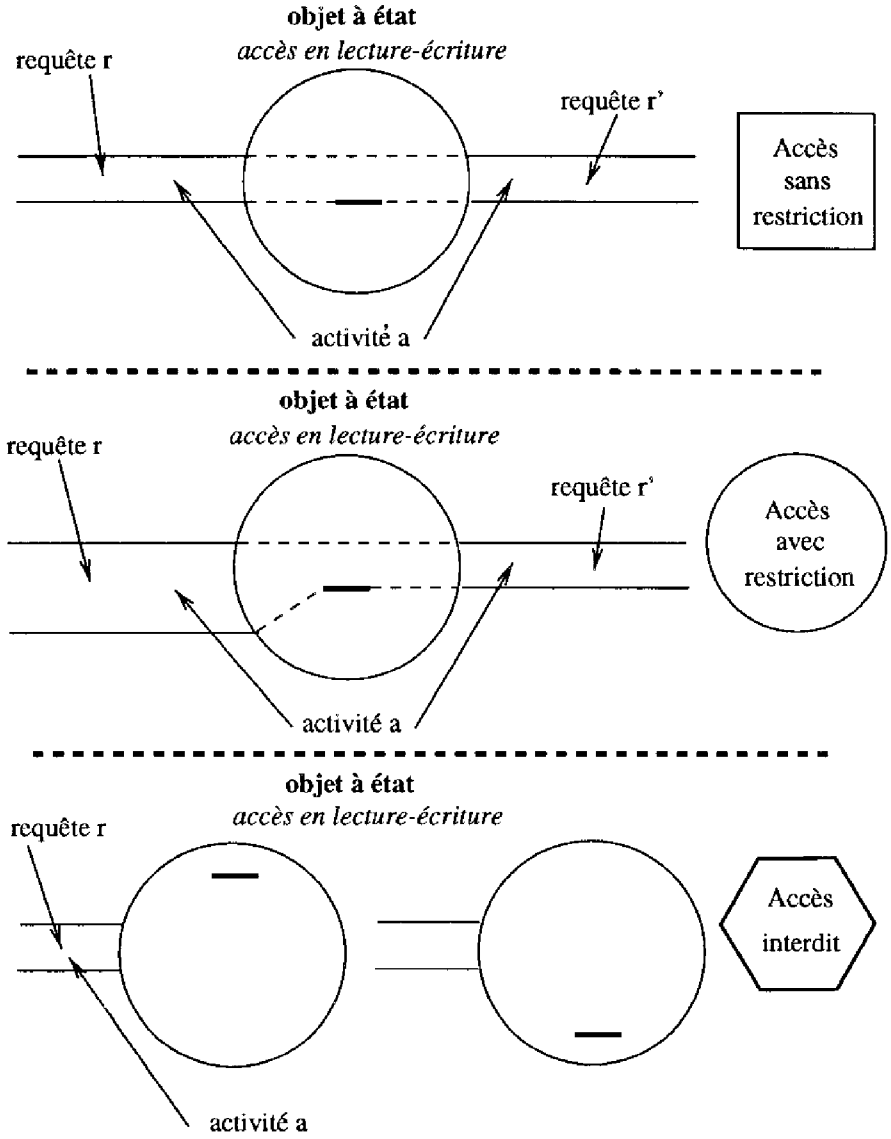


FIG. 3.6 – *Accès en lecture-écriture à un objet à état*

Les retours de fonctions

Il est important de noter que dans toutes ces règles que nous venons de présenter, les restrictions qui sont établies s'appliquent également aux messages qui forment le retour d'un appel de fonction. En effet, la requête r' que nous avons représentée dans chacun des schémas peut très bien être le message formant le retour d'un appel de fonction dont r formait le message de requête.

De plus, un retour de fonction peut également modifier l'état d'un objet et il est donc nécessaire de contrôler les retours de fonction. Si l'on examine d'ailleurs de plus près ce type de flux d'information, nous pouvons nous rendre compte qu'un seul cas est en réalité à considérer : le cas d'un objet à état. En effet, un objet sans état ne conservant pas de mémoire des requêtes y accédant, si l'invocation d'une méthode d'un objet sans état est autorisée (avec ou sans restriction) le retour de cette fonction sera nécessairement autorisé : ce retour n'effectue pas de flux d'information et la parenthèse de labels de la requête effectuant ce retour est forcément incluse dans l'intervalle de confiance de l'objet (cette requête fait partie de l'activité qui a accédé à l'objet et dont la parenthèse a forcément été réduite au plus à l'intervalle de confiance de l'objet). Si l'on considère un objet à état, le retour d'une fonction qu'il a invoquée peut très bien effectuer un flux d'information vers l'état de l'objet et dans ce cas, il est nécessaire de contrôler cette écriture de la même façon que l'appel d'une méthode en écriture de l'objet.

3.5 Exemple

Dans l'exemple qui suit, nous reprenons le schéma développé dans le chapitre 2 mais nous ajoutons aux contrôles discrétionnaires des contrôles obligatoires, à partir des règles que nous venons de présenter.

Nous considérons donc la même opération, qui consiste à imprimer le fichier f_3 sur l'imprimante i_4. Les mêmes objets entrent en jeu dans la réalisation de cette opération, à savoir le serveur de fichier sf_2, le serveur d'impression si_1 et le fichier temporaire ft. Nous utilisons la hiérarchie habituelle de niveaux de sensibilité :

NON-CLASSIFIÉ $<$ CONFIDENTIEL $<$ SECRET $<$ TRÈS-SECRET.

Nous avons de plus :

Utilisateur U : SECRET
$\mathbf{si_1}$: [CONFIDENTIEL, SECRET] (objet sans état)
$\mathbf{fs_2}$: [NON-CLASSIFIÉ, SECRET] (objet sans état)
$\mathbf{i_4}$: [NON-CLASSIFIÉ, CLASSIFIÉ] (objet sans état)
$\mathbf{f_3}$: NON-CLASSIFIÉ (objet à état)

Nous rappelons l'ensemble des interactions composant cet exemple dans la figure 3.7.

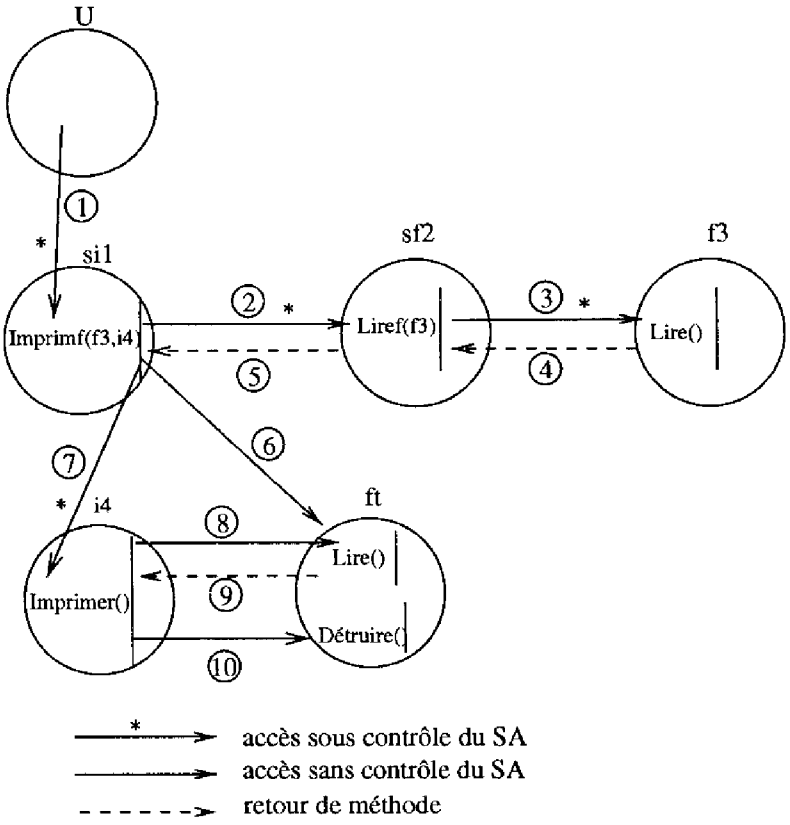


FIG. 3.7 – *Impression d'un fichier*

Le scénario qui représente l'exécution de l'impression est le suivant. Nous ne mentionnons ici que les contrôles obligatoires, sachant qu'à chaque étape, les

contrôles discrétionnaires sont également effectués. Nous ne représentons ici que les contrôles obligatoires pour éviter d'alourdir exagérément le texte.

- L'utilisateur U se connecte sur le système avec l'habilitation SECRET. Une activité a débute alors et effectue la première requête pour U. Cette requête r est, conformément à ce que nous avons expliqué dans le chapitre 2, l'invocation de la méthode $Imprimf$ du serveur d'impression de i_4, soit si_1. Cette requête est étiquetée [NON-CLASSIFIÉ, SECRET] (**message 1**). Le label inférieur est NON-CLASSIFIÉ parce que l'activité en cours ne contient pour le moment aucune information sensible. Il n'y a donc aucune raison de lui attribuer un label supérieur. Le label maximal représente quant à lui, l'habilitation de l'activité, qui est donc initialisée avec celle de l'utilisateur qui l'a déclenchée.

- Le serveur d'impression si_1 est un objet sans état dont l'intervalle de confiance est [CONFIDENTIEL, SECRET]. L'intersection de cet intervalle et de la parenthèse de labels de l'activité est non vide. Mais comme $Lmin_r \prec Lmin_{si_1} \preceq Lmax_r \preceq Lmax_{si_1}$, l'accès est autorisé avec la restriction : $Lmin_a$ devient CONFIDENTIEL (cf. R2).

- L'activité se poursuit en invoquant la méthode $Liref$ du serveur de fichier sf_2 (**message 2**). Cette requête est étiquetée [CONFIDENTIEL, SECRET]. sf_2 est un objet sans état étiqueté [NON-CLASSIFIÉ, SECRET]. L'accès est donc autorisé sans restriction. En effet, ce cas correspond au cas particulier de la règle R5 où la parenthèse de labels de l'activité est incluse dans l'intervalle de confiance de l'objet.

- L'activité invoque alors la méthode $Lire$ du fichier f_3 (**message 3**). Cette requête est étiquetée [CONFIDENTIEL, SECRET]. Le fichier f_3 est un objet à état dont la classification est NON-CLASSIFIÉ et l'accès est un accès en lecture. L'invocation de la méthode est donc autorisée sans restriction (cf. cas particulier de la règle R4).

- L'activité se poursuit en retournant successivement aux méthodes $Liref$ de sf_2 et $Imprimf$ de si_1. Ces deux retours sont étiquetés [CONFIDENTIEL, SECRET] (**messages 4 et 5**). Ils sont autorisés sans restriction parce que leurs parenthèses de labels sont égales à l'intervalle de confiance de sf_2 et si_1 (cas particulier de la règle R2).

- si_1 crée un objet temporaire ft dans lequel il copie f_3 (**message 6**). Cette requête de création fait partie de la même activité et possède donc

la parenthèse de labels [CONFIDENTIEL, SECRET]. Le fichier ft est un objet à état et il est créé avec un label qui est au minimum : $L_{ft} = Lmin_r$. Cette création est accompagné de l'écriture de ft (les données écrites peuvent être passées en paramètre à la création de l'objet).

- L'activité accède alors à la méthode *Imprimer* de l'imprimante i_4 (**message 7**). Cette requête est étiquetée [CONFIDENTIEL, SECRET]. i_4 est un objet sans état et l'intersection de la parenthèse de labels de la requête et de l'intervalle de confiance de i_4 est non vide. Mais, comme $Lmin_{i_4} \preceq Lmin_r \preceq Lmax_{i_4} \prec Lmax_r$. l'accès est autorisé avec la restriction : $Lmax_a$ devient CONFIDENTIEL (cf. R2).

- i_4 invoque la méthode *Lire* du fichier ft (**message 8**). La requête est étiquetée [CONFIDENTIEL, CONFIDENTIEL] (à cause de l'accès précédent à i_4). ft est un objet à état, la méthode invoquée est une méthode en lecture et $L_{ft} = Lmin_r$, aussi l'accès est autorisé sans restriction. Le retour de cette requête (**message 9**) est étiqueté [CONFIDENTIEL, CONFIDENTIEL] et est autorisé parce que cette parenthèse de labels est incluse dans l'intervalle de confiance de si_1. Après l'impression, i_4 détruit le fichier ft (**message 10**) à l'aide d'une requête [CONFIDENTIEL, CONFIDENTIEL]. Cet accès est autorisé sans restriction puisque $Lmin_r = L_{ft}$ (détruire est un accès en écriture).

Dans cet exemple, nous pouvons constater l'intérêt des objets sans état pour une politique de sécurité moins restrictive que celle de Bell-LaPadula. En effet, le label du fichier ft est attribué en fonction du label minimal de la requête qui le crée. Ainsi l'objet si_1 d'habilitation SECRET crée un objet CONFIDENTIEL puisque cette création est réalisée par une requête étiquetée [CONFIDENTIEL, SECRET]. Dans la politique de Bell et LaPadula, le label de l'objet ft est nécessairement supérieur ou égal à celui de si_1. Ainsi, si si_1 est SECRET (un objet possède un seul niveau dans la politique de Bell-LaPadula), le fichier ft est créé SECRET. L'imprimante i_4 ne peut alors plus accéder ce fichier. Il est bien sûr possible d'attribuer un label plus bas à si_1 mais alors il ne pourra plus accéder à des fichiers ayant un label élevé. Ainsi, dans tous les cas de figure, ce type d'opération pose un problème si l'on veut l'adapter à la politique de Bell-LaPadula. Pour pouvoir effectuer les mêmes opérations que le permet notre modèle dans le cadre de la politique de Bell-LaPadula, il faudrait créer plusieurs instances de l'objet si_1 chacune avec un label différent. Chaque instance serait invoquée en fonction de la classification du fichier lu par exemple. De même, il faudrait plusieurs instances de l'imprimante i_4,

chacune étiquetée différemment. On peut se rendre compte qu'un tel système risque de devenir rapidement ingérable. Ainsi notre modèle de sécurité permet bien d'implémenter les mêmes contrôles que la politique de Bell-LaPadula tout en étant moins restrictif que ce modèle.

3.6 Preuve de l'interdiction des flux illégaux

Dans cette partie, nous démontrons que notre modèle permet d'interdire les flux d'information illégaux dans le système.

Interdire les flux d'information illégaux consiste à démontrer qu'il ne doit pas exister de moyen pour un utilisateur d'obtenir de l'information qu'il n'est pas habilité à recevoir, en appliquant les règles du schéma d'autorisation, c'est-à-dire par des canaux légitimes, même avec la collaboration de plusieurs utilisateurs et de plusieurs objets dans le système. Ceci peut être exprimé de la façon suivante :

Nous voulons vérifier que si s est un utilisateur, O un objet à état et $L_O \not\preceq L_s$, alors il n'existe pas de séries $\{i_1, i_2, ..., i_l\}$ et $\{j_1, j_2, ..., j_m\}$ telles que :

$(s_{j_1}, O_{i_1}, lire) \wedge (s_{j_1}, O_{i_2}, écrire) \wedge (s_{j_2}, O_{i_2}, lire) \wedge \cdots \wedge (s_{j_m}, O_{i_l}, lire)$.
avec $s_{j_m} = s$ et $O_{i_1} = O$.

Dans le contexte de notre modèle, ceci signifie qu'il ne doit pas exister de séries d'activités et d'objets dont la collaboration permette des flux d'information illégaux. Formellement, cela signifie qu'il ne doit pas exister de séries $\{i_1, i_2, ..., i_l\}$ et $\{k_1, k_2, ..., k_n\}$ telles que :

$(a_{k_1}, O_{i_1}, lire) \wedge (a_{k_1}, O_{i_2}, écrire) \wedge (a_{k_2}, O_{i_2}, lire) \wedge (a_{k_2}, O_{i_3}, écrire) \wedge \cdots$
$\wedge (a_{k_n}, O_{i_l}, lire)$ ($\forall i, a_{k_i}$ est une activité ; a_{k_n} est intialisée par l'utilisateur s_{j_m})

$(a_{k_1}, O_{i_1}, lire)$ implique :
• $L_{O_{i_1}} \preceq Lmax_{a_{k_1}}$ (Règle R1)
• $Lmin_{a_{k_1}}$ prend la valeur $Max(Lmin_{a_{k_1}}, L_{O_{i_1}})$ (Règle R7)

$(a_{k_1}, O_{i_2}, écrire)$ implique :
. $Lmin_{a_{k_1}} \preceq L_{O_{i_2}}$ (Règle R9).
(avec $Lmin_{a_{k_1}}$ modifié en fonction de l'accès à l'objet O_{i_1}).

Ceci nous amène à: $L_{O_{i_1}} \preceq L_{O_{i_2}}$ et donc de proche en proche à: $L_{O_{i_1}} \preceq L_{O_{i_2}} \cdots \preceq L_{O_{i_l}}$

Mais on a également, $(a_{k_n}, O_{i_l}, lire)$ implique: $L_{O_{i_l}} \preceq Lmax_{a_{k_n}}$. De plus, comme l'activité a_{k_n} est initialisée par s_{j_m}, nous avons la relation: $Lmax_{a_{k_n}} \preceq L_{s_{j_m}}$. Ceci nous amène à: $L_{O_{i_l}} \preceq L_{s_{j_m}}$.

Nous obtenons donc: $L_{O_{i_1}} \preceq L_{O_{i_2}} \cdots \preceq L_{O_{i_l}} \preceq L_{s_{j_m}}$ et donc $L_{O_{i_1}} \preceq L_{s_{j_m}}$, c'est-à-dire $L_O \preceq L_s$ ce qui est contraire à l'hypothèse ($L_O \not\preceq L_s$).

3.7 Processus de validation des objets sans état et des modes d'accès aux objets à état

Un des avantages de notre modèle provient de l'existence des objets sans état et des règles du schéma d'autorisation relatives à ce type d'objet qui permettent des contrôles moins draconiens que le modèle de Bell et LaPadula. Il reste donc à résoudre le problème de la validation dans le système de ce type d'objet. En effet, nous associons à chaque objet sans état un intervalle de confiance. Cet intervalle de confiance est très important car plus il est grand, c'est-à-dire plus on a confiance en l'objet d'être véritablement un objet sans état, et moins la dégradation en terme d'accessibilité de l'information risque d'être importante. Il nous reste donc à trouver des mécanismes qui nous permettent d'attribuer cet intervalle de confiance précisément.

De la même façon, il faut que l'on puisse assurer qu'une méthode d'un objet à état qui prétend être un accès en lecture ne modifie effectivement pas l'état de l'objet. De même, on doit s'assurer qu'une méthode d'écriture ne provoque pas de flux d'information vers l'activité qui l'invoque.

Nous pensons que ces vérifications peuvent être effectuées sans trop de difficultés grâce à certaines techniques classiques telles que l'analyse de programmes ou la preuve de programme. En effet, les propriétés que nous voulons vérifier sont relativement simples. Pour prouver qu'un objet prétendu sans état est réellement un objet sans état, il suffit de vérifier qu'aucune des méthodes de cet objet ne garde mémoire des différentes informations échangées avec les activités qui y accèdent. Cette vérification peut s'effectuer en s'assurant que toutes les variables locales des méthodes sont réinitialisées à l'invocation et qu'il n'y a aucune interaction entre les variables globales de l'objet et les informations contenues dans la requête qui y accède. De même, pour véri-

fier qu'une méthode prétendue être un accès en lecture est bien un accès en lecture, il suffit de s'assurer qu'aucune variable (locale ou globale) de l'objet n'est modifiée lors de l'invocation de la méthode.

Dans le cas où il ne serait pas possible d'avoir confiance dans ces vérifications, par exemple parce qu'il n'est pas possible d'obtenir le code source des objets correspondants, nous pouvons toujours opter pour la position de repli qui consiste à considérer que tous les objets sont des objets à état et que toutes leurs méthodes sont des méthodes de lecture-écriture.

3.8 Les canaux cachés

3.8.1 Le but recherché : la validation des objets

Comme nous l'avons indiqué dans le paragraphe 3.1.3.2.1, l'élimination des canaux cachés dans un système est une tâche quasiment impossible. On préfère souvent limiter la bande passante de ces canaux cachés. Nous choisissons dans notre cas, d'analyser le code de chaque objet que nous introduisons dans notre système de façon à déterminer si celui-ci tente d'utiliser un canal caché pour transmettre de l'information. Cette analyse nous semble réalisable et efficace. En effet, pour valider un système au niveau A1, le livre orange exige d'identifier tous les canaux cachés que contient un tel système. Or des systèmes ont été évalués A1, ce qui signifie bien que cette étude exhaustive peut être réalisée. Aussi, nous pensons que, a fortiori, il est possible d'analyser chacun des objets de notre système de façon à déterminer s'il est susceptible d'utiliser un canal caché ou pas. Plutôt que d'éliminer les canaux cachés, il s'agit ici d'en détecter l'utilisation. Il nous semble que des moyens actuels permettant de réaliser cette analyse sont à notre disposition et que l'avantage d'une telle approche est qu'elle ne grève pas les performances du système. Pour appuyer ces propos, nous décrivons brièvement dans la suite trois méthodes qui permettent d'identifier les canaux cachés dans un système et nous étudions si l'utilisation de ces méthodes peut permettre de valider des objets comme n'utilisant pas de canaux cachés.

3.8.2 La matrice de Kemmerer

Kemmerer est l'un des premiers à avoir proposé une méthode systématique pour identifier des canaux cachés dans un système [Kem82]. Cette méthode permet d'identifier un grand nombre de canaux cachés de mémoire et éventuellement certains canaux cachés temporels. Elle consiste tout d'abord à rechercher toutes les variables partagées ainsi que toutes les primitives permettant d'y accéder (que ce soit en lecture ou en écriture), puis à construire une matrice dont chaque ligne est une variable partagée et chaque colonne une primitive. Si une primitive modifie une variable partagée alors la valeur "M" sera contenue dans la matrice à l'intersection de la ligne correspondant à la variable et de la colonne correspondant à la primitive. Il en est de même avec la valeur "R" pour une lecture. Toute ligne qui contient une modification et une lecture est source d'un canal caché potentiel : la variable correspondant à cette ligne peut être consultée par une primitive et modifiée par une autre. L'intérêt d'une telle méthode est qu'elle est applicable aussi bien à une description informelle, des spécifications formelles ou du code. Nous donnons ci-dessous l'exemple d'une telle matrice appliquée à une séquence de code.

```
procedure WriteFile(fileid:filerange);
begin
  if (files[fileid].locked) and (files[fileid].owner=currentprocess))
    then files[fileid].value:=processes[currentprocess].buffer
end;
```

La matrice produite est représentée ci-après :

Variables/Primitives	*WriteFile*	*ReadFile*	...
process.id		...	...
process.accessrights		...	...
process.buffer	R	...	...
file.value	M	...	...
file.owner	R	...	...
file.locked	R	...	...

Dans le cadre de notre travail de validation des objets, cette méthode nous permet d'identifier les variables grâce auxquelles des canaux cachés pourraient être exploités. Ceci nous donne déjà un bon nombre de renseignements précieux dans la recherche d'utilisation de ces canaux cachés. De plus, la souplesse

de cette méthode est intéressante puisqu'elle est applicable au niveau du codage mais aussi des spécifications. Néanmoins, cette méthode est insuffisante car elle permet de dégager des variables dont l'utilisation pourrait mener à des flux d'information par canaux cachés mais elle ne permet pas d'établir des scénarios réalisant effectivement ces flux. Ainsi que le dit Kemmerer lui-même, pour identifier les canaux cachés grâce à sa méthode, il faut faire preuve d'un minimum d'imagination pour envisager les scénarios possibles à partir des variables identifiées.

3.8.3 L'arbre des flux

La méthode de l'arbre des flux, mise au point par Porras et Kemmerer [PK91] utilise une méthode déductive pour obtenir non seulement les variables du système susceptibles d'être utilisées pour créer un canal caché mais aussi les scénarios qui peuvent mener à ces flux illégaux. Un arbre des flux est construit, représentant un certain nombre de chemins possibles permettant de réaliser un même canal caché (c'est-à-dire un canal caché basé sur l'utilisation d'une variable particulière).

L'algorithme de création de l'arbre des flux nécessite une base de données associée à chaque primitive. Ainsi, il est nécessaire d'établir la liste des variables dont les valeurs sont référencées lors de l'exécution de chaque primitive ainsi que la liste des variables dont la valeur est modifiée et la liste des variables dont la valeur est retournée. Un arbre est associé à chaque cible de recherche qui constitue sa racine : par exemple, l'identification du canal caché dû à l'utilisation d'une variable donnée A. La figure 3.8 présente un exemple d'un tel arbre.

Cet exemple représente un chemin conduisant à l'exploitation d'un canal caché basé sur l'utilisation de la valeur de la variable *Ouvert*. Le chemin représenté représente un cas très simple où l'émetteur ouvre ou ferme un fichier et le récepteur utilise la primitive `Fichier_Ouvert` pour détecter si le fichier est ouvert ou fermé. Si l'émetteur et le récepteur se sont mis d'accord sur le code : "fichier ouvert = 0 et fichier fermé = 1", une chaîne de bits peut être transmise ainsi.

Dans le cadre de ce mémoire, cette méthode nous intéresse davantage que la précédente puisqu'elle est capable de nous fournir des scénarios d'enchaînement de primitives qui conduisent à l'exploitation d'un canal caché. Ces scénarios étant obtenus, il est alors relativement facile de vérifier qu'un objet

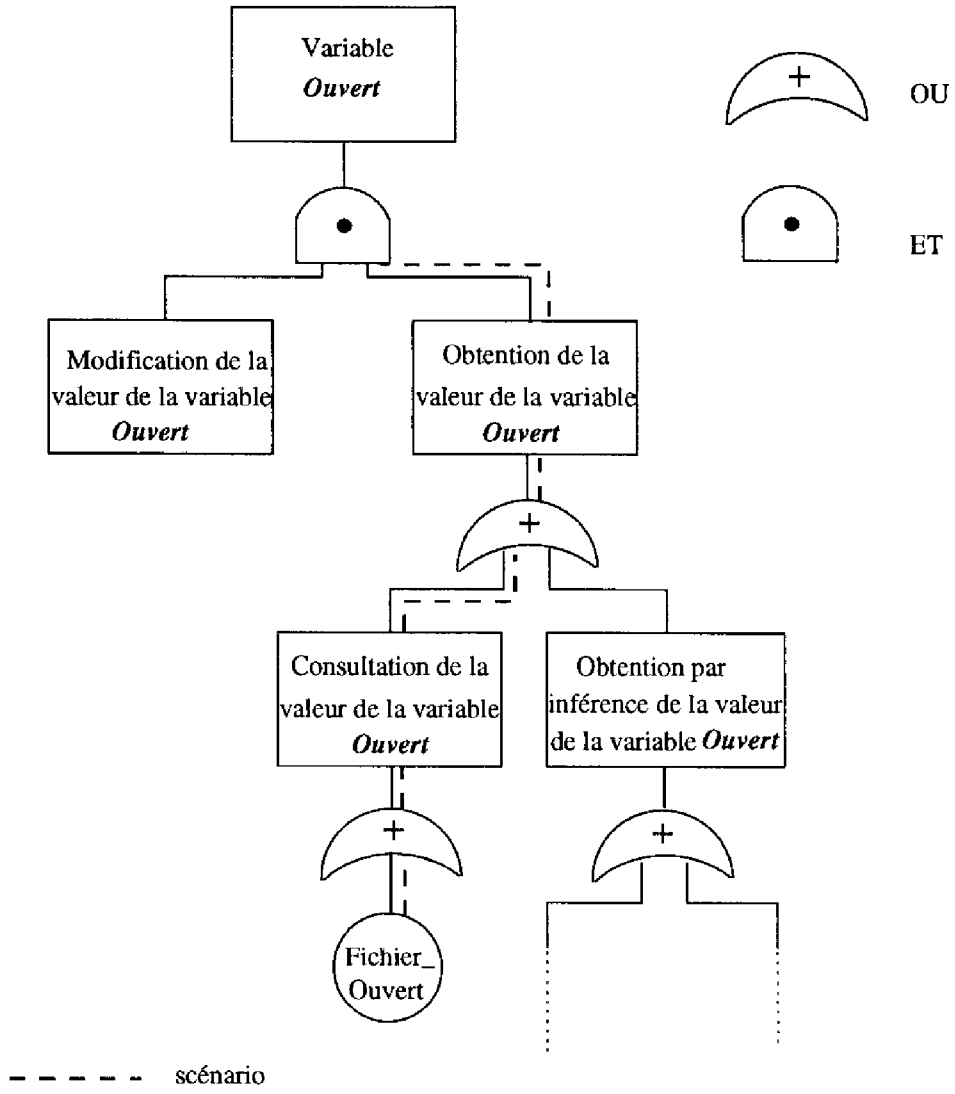


FIG. 3.8 – *Un exemple de scénario de canal caché*

ne les exploite pas.

Citons enfin les travaux menés par Yves Bergeon dans [Ber96] où un outil d'aide à la détection des canaux cachés dans les protocoles permet l'identifi-

cation de canaux cachés dans une spécification écrite en langage Estelle.

3.8.4 L'analyse du code C

Le modèle précédent s'applique typiquement à des spécifications d'un système. Or, l'implémentation de spécifications sur une architecture précise dans un langage précis peut rajouter des canaux cachés dus à la structure du langage lui-même. Aussi, il est intéressant de réaliser une identification des canaux cachés dans le code source même des programmes. Plus le langage est proche du langage machine, et plus l'analyse est précise. C'est ainsi que Tsai, Gligor et Chandersekaran ont développé une méthode permettant d'identifier des canaux cachés dans des programmes en langage C. Dans [TGC90], ils détaillent cette méthode et l'expérimentent sur le noyau de Secure Xenix. La méthode employée peut se résumer ainsi :

- sélection des primitives du noyau directement visibles pour un utilisateur ;
- détermination des variables directement et indirectement altérables et visibles du noyau par :
  - détermination des variables du noyau directement altérables et visibles par analyse de la sémantique du langage pour chaque primitive sélectionnée ;
  - détermination des dépendances des appels de fonctions (*function-call dependency*) et des variables visibles et altérables par chaque primitive ;
  - détermination des variables indirectement visibles par analyse des flux d'information ;
  - élimination des variables locales pour déterminer exactement la liste des variables partagées entre plusieurs primitives.
- analyse des variables partagées et identification des canaux cachés.

3.8.5 Discussion sur les canaux cachés

Les méthodes actuelles permettent d'identifier des canaux cachés dans les systèmes informatiques à deux niveaux : spécification et codage. Notre propos

n'est pas d'identifier les canaux cachés mais de détecter l'utilisation éventuelle de canaux cachés par les objets introduits dans notre système :

- Comme nous venons de le montrer, les méthodes qui permettent d'identifier les canaux cachés peuvent être un support très utile pour détecter l'utilisation de canaux cachés. Par exemple, l'analyse du code C grâce à la méthode de Tsai, Gligor et Chandersekaran permet de mettre en évidence un certain nombre de variables pouvant être exploitées pour créer des canaux cachés. Ces renseignements nous permettent d'orienter la recherche de l'utilisation de canaux cachés.

- En dehors même des méthodes permettant d'identifier les canaux cachés dans un systèmes, ce qui est finalement un problème bien plus vaste que le problème qui nous intéresse, il nous semble vraisemblable qu'une analyse détaillée du code d'un objet peut assez facilement révéler des tentatives d'utilisation de canaux cachés. En effet, l'utilisation d'un canal caché dans un programme donne lieu à un code qui n'a aucun rapport avec la fonction de l'objet. Aussi, si on connaît précisément la fonction d'un objet, il semble vraisemblable de pouvoir détecter des parties douteuses de code ne concernant en rien cette fonction. Par exemple, il semble difficile de penser qu'un objet essayant de moduler son temps d'utilisation des ressources puisse le faire sans que le code résultant de cette opération ne le révèle.

3.9 Conclusion

Nous avons présenté dans de chapitre un modèle de sécurité multiniveau adapté au modèle objet. Ce modèle présente l'avantage d'être moins restrictif que le modèle de Bell et LaPadula, grâce à l'utilisation des objets sans état. Il peut de plus s'intégrer dans le schéma d'autorisation que nous avons présenté dans le chapitre précédent pour ajouter des contrôles obligatoires aux classiques contrôles discrétionnaires.

Chapitre 4

Exemple d'implémentation du schéma d'autorisation sur une architecture à base de micro-noyaux

Ce chapitre est consacré à la présentation d'une implémentation du schéma d'autorisation. Cette implémentation est réalisée sur un ensemble de machines équipées du micro-noyau CHORUS. Nous présentons dans une première partie l'exemple qui a été choisi pour cette implémentation et nous détaillons dans une seconde partie la façon dont nous avons implémenté cet exemple sur le micro-noyau, en essayant de présenter les choix que nous avons faits ainsi que les raisons de ces choix.

4.1 Présentation de l'exemple

4.1.1 Introduction

L'exemple que nous avons choisi pour illustrer les principes de notre schéma d'autorisation est celui de la messagerie électronique. Nous allons détailler l'envoi d'un courrier d'un utilisateur à un autre utilisateur. Cet exemple nécessite trois opérations de haut niveau, dont nous détaillerons ici la plus im-

portante. Nous invitons le lecteur à se référer aux annexes pour trouver la présentation des deux autres opérations.

Dans l'exemple que nous avons conçu, l'utilisateur u désire envoyer un courrier électronique à l'utilisateur v. L'utilisateur u commence tout d'abord par éditer un fichier dans lequel il écrit son message (cette opération ne sera pas détaillée ici). Il demande ensuite à envoyer un courrier électronique à l'utilisateur v, le courrier étant constitué du fichier qu'il vient d'écrire (nous détaillons dans ce chapitre cette opération). L'utilisateur v enfin demande la lecture de son courrier électronique et consulte ainsi le message de l'utilisateur u (cette opération n'est pas détaillée ici).

Nous commençons tout d'abord par présenter les différents objets qui interviennent dans la réalisation de l'opération "envoyer un courrier électronique à un utilisateur à partir d'un fichier". Nous détaillons ensuite les règles qui interviennent pour effectuer les contrôles discrétionnaires de notre schéma d'autorisation. Nous finissons en détaillant les contrôles obligatoires qui sont effectués lors de la réalisation de cette opération. Dans la suite de ce chapitre, un "courrier" désignera toujours un "courrier électronique".

4.1.2 Les objets et leurs méthodes

Les différents objets qui interviennent dans la réalisation de l'opération de haut niveau "envoyer un courrier à un utilisateur à partir d'un fichier" sont les suivants :

- l'interpréteur de commandes qui permet à l'utilisateur émetteur du courrier de se connecter et de demander au système la réalisation de cette opération ; la classe de cet objet est la classe SHELL ;
- des serveurs de fichiers utilisés pour tous les accès aux fichiers ; la classe de ces objets est la classe SERVFICH ; notons que deux objets de cette classe sont nécessaires pour réaliser l'opération : sur le site d'où est émis le courrier, le serveur de fichier gère le fichier destiné à être le contenu du courrier, sur le site recevant le courrier, le serveur de fichier gère le fichier dans lequel va être stocké le courrier ;
- un fichier contenant le courrier à émettre et un fichier contenant le courrier reçu ; la classe de ces objets est la classe FICHIER ;

- les objets chargés de transmettre les courriers d'une machine à une autre ; la classe de ces objets est la classe MTA (tiré de l'anglais "Mail Transport Agent") ; notons que pour la réalisation de l'opération de haut niveau, au moins deux objets de ce type sont nécessaires : un sur la machine émettant le courrier et un sur la machine recevant le courrier ;
- l'objet chargé de déposer le courrier proprement dit ; la classe de cet objet est la classe MDA (tiré de l'anglais "Mail Delivery Agent") ;
- un objet temporaire chargé d'indiquer localement si le courrier envoyé correspond bien à une adresse valable ; la classe de cet objet est la classe MESSAGE.

Nous présentons maintenant les différentes classes de ces objets associées à leurs méthodes :

```
SERVFICH : Liref(FICHIER) ; Écriref(FICHIER) ;
FICHIER  : Lire() ; Écrire() ;
MTA      : Envoyer_Courrier(FICHIER , user) ;
           Recevoir_Courrier(char * , user) ;
MDA      : Livrer_Courrier(FICHIER) ;
MESSAGE  : Afficher(char *) ;
```

Notons le cas particulier de l'objet de classe SHELL qui n'est utilisé que pour invoquer les méthodes des autres objets et qui lui, par conséquent, ne possède pas à proprement parler de méthodes à invoquer, il est constamment actif en attente de requêtes de l'utilisateur.

La figure 4.1 représente les diverses opérations qui interviennent dans la réalisation de l'opération de haut niveau : "l'utilisateur u envoie un courrier à l'utilisateur v à partir du fichier f".

Ce scénario correspond aux opérations suivantes :

1. l'objet *sh* invoque la méthode *Envoyer_Courrier* de l'objet de *mta1* avec les paramètres f et v ;
2. l'objet *mta1* invoque la méthode *Liref* du serveur de fichier *sf1* avec le paramètre f ;

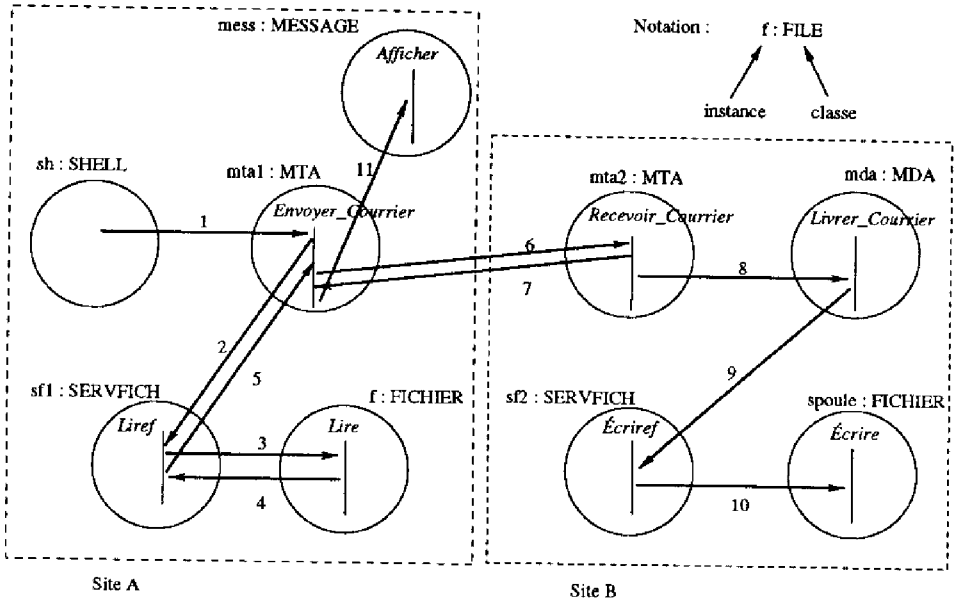


FIG. 4.1 – *Envoi d'un courrier électronique*

3. le serveur de fichier *sf1* invoque la méthode *Lire* du fichier f ;
4. le contenu du fichier f est transmis à *sf1* ;
5. le contenu du fichier f est transmis à *mta1* ;
6. l'objet *mta1* invoque la méthode *Recevoir_Courrier* de l'objet *mta2* avec le paramètre v et le contenu du fichier f ;
7. l'objet *mta2* détermine le fichier de spoule de l'utilisateur v (*spoule*) et renvoie un message à *mta1* si l'utilisateur en question possède bien une boîte aux lettres locale ;
8. l'objet *mta2* invoque la méthode *Livrer_Courrier* de l'objet *mda* avec comme paramètre le fichier *spoule* ;
9. l'objet *mda* forme le message reçu de façon à l'écrire au format du courrier électronique et invoque ensuite la méthode *Écriref* du serveur de fichier *sf2* avec comme paramètre le message ainsi formé ;
10. le serveur de fichier *sf2* invoque la méthode *Écrire* du fichier *spoule* avec comme paramètre le message formé par *mda* ;

11. parallèlement aux opérations 8, 9 et 10, l'objet *mta1* sur le site A crée un objet temporaire *mess* dans le but d'avertir l'utilisateur u si l'adresse qu'il a utilisée est bien valide ou non.

4.1.3 Les contrôles discrétionnaires

Cette section est consacrée à la présentation des contrôles effectués lors de cette opération de haut niveau et en particulier, à la présentation des différentes règles utilisées par le schéma d'autorisation.

4.1.3.1 Les règles de droits symboliques

Plusieurs opérations de haut niveau sont réalisées dans ce schéma. La première correspond à la requête de l'utilisateur u. Celui-ci demande à envoyer un courrier à l'utilisateur v à partir d'un fichier f . L'opération de haut niveau correspondante est notée: **envoyercourrier(f,ut)** (où *ut* désigne un utilisateur quelconque). La lecture d'un fichier et l'écriture d'un fichier sont des opérations de haut niveau et sont respectivement notées: **lirefichier(f)** et **écrirefichier(f)**. Enfin l'écriture locale d'un courrier par l'objet *mta2* correspond à une opération de haut niveau notée: **livrercourrier(f)**.

Les règles correspondant à ces opérations de haut niveau sont les suivantes:

$$RS1(e, envoyercourrier, EC, f, ut) : \frac{\mathrm{EC}_{\mathtt{e},\mathtt{f}}(\mathtt{f},\mathtt{ut}), \mathrm{EC}_{\mathtt{e},\mathtt{ut}}(\mathtt{f},\mathtt{ut})}{\mathbf{A(envoyercourrier)(e,f,ut)}}$$

$$RS2(e, livrercourrier, LC, f) : \frac{\mathrm{LC}_{\mathtt{e},\mathtt{f}}(\mathtt{f})}{\mathbf{A(livrercourrier)(e,f)}}$$

$$RS3(e, écrirefichier, EF, f) : \frac{\mathrm{EF}_{\mathtt{e},\mathtt{f}}(\mathtt{f})}{\mathbf{A(écrirefichier)(e,f)}}$$

$$RS4(e, lirefichier, LF, f) : \frac{\mathrm{LF}_{\mathtt{e},\mathtt{f}}(\mathtt{f})}{\mathbf{A(lirefichier)(e,f)}}$$

La règle RS1 signifie que l'entité *e* est autorisée à réaliser l'opération **envoyercourrier(f,ut)** si cette entité possède le droit symbolique `EC` pour le fichier f et pour une liste d'autres arguments qui comprend *ut* et si elle possède le droit symbolique `EC` pour l'utilisateur *ut* et une liste d'autres arguments qui comprend *f*. Les règles SR2, SR3 et SR4 sont plus simples. La règle SR2 par exemple signifie que l'entité *e* est autorisée à réaliser l'opération *livrercourrier*(*f*) si elle possède le droit symbolique `LC` pour le fichier *f*. Les règles SR3 et SR4 s'interprètent de la même façon.

Le tableau suivant représente un exemple de matrice de droits d'accès qui autorise l'utilisateur *u* à réaliser l'opération **envoyercourrier(f,v)**, qui autorise l'objet *mta1* à réaliser l'opération **lirefichier(f)**, qui autorise l'objet *mta2* à réaliser l'opération **livrercourrier(spoule)** et qui autorise l'objet *mda* à réaliser l'opération **écrirefichier(spoule)**.

	f	v	spoule	...
u	`EC(this,rôleA)`	`EC(FICHIER,this)`		...
mta1	`LF(this)`			...
mta2			`LC(this)`	...
mda			`EF(this)`	...

Le rôle *rôleA* représente ici un des rôles de l'utilisateur *v*. Ainsi le droit symbolique `EC(this,rôleA)` situé à l'intersection de la ligne de utilisateur *u* et de la colonne du fichier *f* signifie que l'utilisateur *u* est autorisé à envoyer un courrier à partir du fichier *f* à n'importe quel utilisateur ayant le rôle *rôleA*.

Notons qu'un objet peut être autorisé à effectuer une opération de haut niveau sans pour autant posséder dans la matrice les droits requis par la règle de droits symboliques correspondante. En effet, ainsi que nous l'avons expliqué dans le chapitre 2, lorsqu'un objet est autorisé à effectuer une opération de haut niveau, il reçoit une capacité lui permettant de débuter cette opération et il reçoit également un certain nombre de coupons. Un coupon peut être une capacité ou l'autorisation de réaliser une autre opération de haut niveau. Ainsi, lorsqu'un objet va demander à réaliser une opération de haut niveau, il peut présenter au serveur d'autorisation un coupon contenant cette autorisation alors qu'il ne possède pas les droits symboliques qui sont requis par la règle de droits symboliques correspondante dans la matrice des droits d'accès. Ainsi la matrice des droits d'accès présentée ci-dessus est une matrice suffisante mais pas nécessaire comme nous le verrons dans la suite.

4.1.3.2 Les règles de création de capacités

Ces règles indiquent les capacités et les coupons qui sont délivrés à un objet autorisé à réaliser l'une des opérations de haut niveau ou une opération élémentaire.

Opérations de haut niveau :

(RC1) $\forall f$ *instance de* $FICHIER$, $\forall ut$ *utilisateur*,
$Cap(\mathbf{o}, \mathbf{envoyercourrier(f, ut)}) ::=$
$\{Cap(o, !MTA.Envoyer\_Courrier(f, ut))$, $[\mathbf{A(lirefichier)}(!MTA, \mathbf{f})]\}$

La règle RC1 signifie que lorsqu'un objet est autorisé à réaliser l'opération **envoyercourrier(f,ut)**, il reçoit une capacité correspondant à la méthode *Envoyer_Courrier* de l'objet local de classe MTA [10] ; il reçoit également un coupon qui correspond à l'autorisation de réaliser l'opération de haut niveau **lirefichier(f)** pour l'instance locale de la classe MTA.

(RC2) $\forall f$ *instance de* $FICHIER$,
$Cap(\mathbf{o}, \mathbf{livrercourrier(f)}) ::=$
$\{Cap(o, !MDA.Livrer\_Courrier(f))$,
$[\mathbf{A(écrirefichier)}(\mathsf{ServeurFich}(\mathbf{f}), \mathbf{f})]\}$

La règle RC2 signifie que lorsqu'un objet est autorisé à réaliser l'opération **livrercourrier(f)**, il reçoit la capacité lui permettant d'accéder à la méthode *Livrer_Courrier* de l'instance locale de la classe MDA et un coupon correspondant à l'autorisation d'effectuer l'opération de haut niveau **écrirefichier(f)** pour l'instance locale de la classe MDA.

(RC3) $\forall f$ *instance de* $FICHIER$,
$Cap(\mathbf{o}, \mathbf{écrirefichier(f)}) ::=$
$\{Cap(o, \mathsf{ServeurFich}(f).\acute{E}crire f(f))$, $[Cap(\mathsf{ServeurFich}(f), f.\acute{E}crire())]\}$

La règle RC3 signifie que lorsqu'un objet est autorisé à réaliser l'opération de haut niveau **écrirefichier(f)**, il reçoit la capacité lui permettant d'accéder à la méthode *Écriref* du serveur de fichiers de f et un coupon qui est la capacité pour le serveur de fichier de f d'accéder à la méthode *Écrire* du fichier f.

10. La notation !MTA représente l'instance locale de la classe MTA, cette notation suppose l'existence d'une instance locale unique de la classe.

(RC4) $\forall f$ *instance de* $FICHIER$,
$Cap(\mathbf{o}, \mathbf{lirefichier(f)}) ::=$
$\{Cap(o, \mathsf{ServeurFich}(f).Liref(f)), [Cap(\mathsf{ServeurFich}(f), f.Lire())]\}$

La règle RC4 est la règle duale de la règle RC3 pour la lecture.

Opérations élémentaires :

Toutes les règles de création de capacités dans le cas d'opérations élémentaires étant similaires, nous donnons simplement un exemple.

(RC5) $\forall f$ *instance de* $FICHIER$,
$Cap(o, f.Lire()) ::= \{ref_o, ref_{f.Lire}, nonce\}$

Compte tenu de ces règles, le paragraphe suivant présente le scénario complet incluant les contrôles effectués par le serveur d'autorisation et les noyaux de sécurité ainsi que la matrice des droits d'accès utilisée.

4.1.3.3 Le scénario

Étant donné la délégation des droits qui est effectuée par l'intermédiaire des coupons, la matrice d'accès suffisante est donnée dans le tableau suivant :

	f	v	spoule	mta2
u	EC(this,rôleA)	EC(FICHIER,this)		
mta1				Recevoir_Courrier
mta2			LC(this)	

Le scénario est alors le suivant (les numéros de messages se rapportent à la figure 4.1 :

- L'objet *sh* demande l'autorisation pour l'utilisateur *u* de réaliser l'opération de haut niveau **envoyercourrier(f,v)**. La matrice des droits d'accès et la règle de droits symboliques SR1 autorisent cet accès. Conformément à la règle RC1, le serveur d'autorisation fournit à l'objet *sh*, la capacité lui permettant d'accéder à la méthode *Envoyer_Courrier* de *mta1* et un coupon autorisant *mta1* à effectuer l'opération de haut niveau **lirefichier(f)**.
- L'objet *sh* invoque la méthode *Envoyer_Courrier* (**message 1**) de l'objet *mta1*. Cette invocation est contrôlée par le noyau de sécurité local qui

vérifie que le message est bien accompagné de la capacité autorisant l'opération[11]. L'objet *sh* fournit à *mta1* le coupon lui permettant de réaliser l'opération de haut niveau **lirefichier(f)**.

- L'objet *mta1* demande à réaliser l'opération de haut niveau **lirefichier(f)**. Il transmet cette demande au SA en lui présentant le coupon reçu de l'objet *sh*. Ainsi, bien que *mta1* ne possède pas les droits symboliques requis par la règle de droits symboliques RS4, il est autorisé à réaliser cette opération grâce au coupon. Il reçoit, conformément à la règle RC4, la capacité lui permettant d'accéder à la méthode *Liref* de *sf1* et un coupon contenant la capacité pour *sf1* d'accéder à la méthode *Lire* de f.

- L'objet *mta1* accède à la méthode *Liref* de *sf1* (**message 2**) et lui transmet le coupon lui permettant d'accéder à la méthode *Lire* de *f*.

- Le serveur de fichier *sf1* accède à la méthode *Écrire* du fichier *f* (**message 3**) grâce au coupon reçu de *mta1*. Il n'a donc pas à demander au serveur d'autorisation une capacité autorisant cet accès.

- Le contenu du fichier *f* revient à *mta1* par l'intermédiaire de deux retours de fonctions (**messages 4 et 5**).

- L'objet *mta1* demande au serveur d'autorisation le droit d'invoquer la méthode *Recevoir_Courrier* de l'objet *mta2* (cet accès est un accès élémentaire). Le serveur d'autorisation consulte la matrice des droits d'accès et forme ensuite la capacité permettant à *mta1* d'effectuer cette opération.

- L'objet *mta1* accède à la méthode *Recevoir_Courrier* de l'objet *mta2* (**message 6**).

- L'objet *mta2* vérifie que l'utilisateur *v* possède bien une boîte aux lettres localement et renvoie un message indiquant le résultat de cette recherche à *mta1* (**message 7**).

- L'objet *mta2* demande au serveur d'autorisation l'autorisation d'effectuer l'opération de haut niveau **livrercourrier(spoule)**. Le serveur d'autorisation vérifie que cet accès est autorisé conformément à la règle

11. Dans la suite de la présentation de ce scénario, nous ne mentionnerons plus les contrôles effectués par le noyau de sécurité, sachant qu'ils sont effectués à chaque accès à un objet local.

de droits symboliques RS2 et fournit à *mta2*, conformément à la règle de création de capacités RC2, la capacité lui permettant d'accéder à la méthode *Livrer_Courrier* de l'objet *mda* et un coupon correspondant à l'autorisation pour *mda* d'effectuer l'opération de haut niveau **écrirefichier(spoule)**.

- L'objet *mta2* accède à la méthode *Livrer_Courrier* de *mda* (**message 8**) et lui transmet le coupon reçu du serveur d'autorisation.

- L'objet *mda* demande à réaliser l'opération **écrirefichier(spoule)**. Il adresse cette requête au serveur d'autorisation en y joignant le coupon reçu de *mta2*. Le serveur d'autorisation autorise donc la réalisation de l'opération à la lecture du coupon. Il fournit à *mda*, conformément à la règle RC3, la capacité permettant d'accéder à la méthode *Écriref* de *sf2* et un coupon contenant la capacité permettant à *sf2* d'accéder à la méthode *Écrire* de *spoule*.

- L'objet *mda* accède à la méthode *Écriref* de *sf2* (**message 9**) et transmet à *sf2* le coupon reçu du serveur d'autorisation.

- L'objet *sf2* accède à la méthode *Écrire* de *spoule* (**message 10**) grâce au coupon reçu de *mda*.

- L'objet *mta1*, pour indiquer à l'utilisateur *u* si l'adresse qu'il a utilisée est valide crée un objet temporaire de classe MESSAGE. Cette création est contrôlée par le noyau de sécurité local uniquement. L'objet *mta1* invoque ensuite la méthode *Afficher* de cet objet (**message 11**), cet accès étant également contrôlé par le noyau de sécurité local.

4.1.4 Les contrôles obligatoires

Nous nous limitons dans cette partie à donner un exemple d'étiquetage des objets et de représenter par une figure l'évolution des labels au cours de la requête.

Un exemple d'étiquetage est représenté sur la figure 4.2.

mta1 : [CONFIDENTIEL, TRÈS-SECRET];
sf1 : [NON-CLASSIFIÉ, SECRET];
mta2 : [CONFIDENTIEL, TRÈS-SECRET];

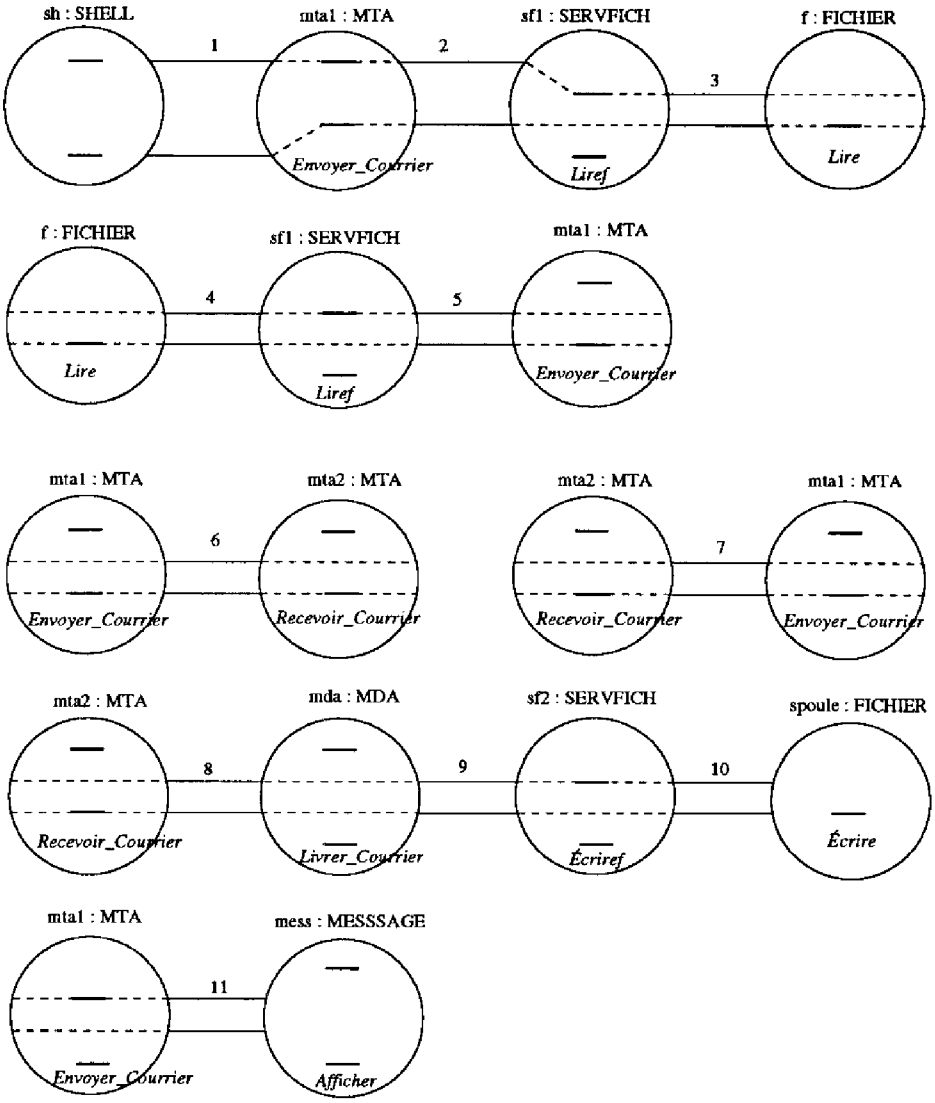


FIG. 4.2 – *Exemple d'étiquetage*

mda : [NON-CLASSIFIÉ, TRÈS-SECRET];
sf2 : [NON-CLASSIFIÉ, SECRET];
mess : [NON-CLASSIFIÉ, TRÈS-SECRET].

Notons que la classification du courrier envoyé dépend de la confiance que l'on accorde aux objets *mta1* et *sf1*. Ainsi dans cet exemple, la confiance que l'on accorde à l'objet *mta1* est limitée de telle sorte qu'on l'empêche de manipuler un courrier non-classifié. En conséquence, l'émetteur du courrier ne peut envoyer de message non-classifié. Ceci reste en total accord avec notre schéma d'autorisation : plus on valide scrupuleusement les objets sans état (c'est-à-dire plus on est certain que ces objets sont réellement des objets sans état), plus on peut leur attribuer un intervalle de confiance large et plus la liberté d'exécution d'opérations est grande. Notons que, dans cet exemple, l'utilisateur émettant le courrier peut envoyer un courrier dont la classification domine l'habilitation de l'utilisateur destinataire du courrier. Il n'y aura pas de flux d'information illégaux car le destinataire du courrier ne pourra le lire mais cela provoque néanmoins un stockage inutile de courrier. Une solution à ce problème est d'instancier un objet sans état dont le label supérieur est initialisé avec l'habilitation du destinataire du courrier. Cet objet est un passage obligé pour poster le courrier si bien qu'un message dominant l'habilitation du destinataire (et donc du label supérieur de cet objet) sera refusé. On peut penser par exemple à instancier un objet de classe MDA spécifique à un utilisateur donné.

4.2 Implémentation de l'exemple sur une architecture à base de micro-noyaux

Cette partie est destinée à la présentation des principaux choix que nous avons faits lors de l'implémentation de notre prototype de noyau de sécurité sur le micro-noyau CHORUS.

4.2.1 Présentation du micro-noyau CHORUS

Notre but n'est en aucun cas ici de présenter une étude exhaustive des caractéristiques du micro-noyau CHORUS. Nous invitons le lecteur à consulter [CHO93, CHO94]. Nous rappelons brièvement ce qu'est un micro-noyau et les principales caractéristiques du noyau CHORUS que nous avons utilisées.

Un micro-noyau est un noyau de système de très petite taille, ne possédant pas toutes les caractéristiques des noyaux monolithiques tels qu'Unix et qui est donc précieux au sens où il constitue une brique de base pour construire

de multiples systèmes d'exploitation. Le micro-noyau CHORUS comprend plusieurs fonctionnalités importantes telles que :

- la gestion de la mémoire (linéaire, segmentée ou paginée) ;
- la gestion des processus légers ou "**threads**" ;
- la gestion des communications (ou IPC : Inter-Process Communication) ;
- la gestion des interruptions matérielles et logicielles.

Ces fonctionnalités essentielles font que ce type de micro-noyau fournit un service suffisant pour permettre de développer des systèmes d'exploitation sans avoir à tout construire. D'un autre côté, ce type de micro-noyau est suffisamment simple et adaptable pour permettre de construire de multiples systèmes. Par exemple, la gestion des fichiers ne fait pas partie intégrante du noyau, ce qui laisse la possibilité à un utilisateur de définir sa propre gestion de fichiers en construisant un "sous-système" personnel au dessus du micro-noyau.

Construire un sous-système au dessus du micro-noyau consiste à concevoir et développer certaines fonctionnalités requises et à les exécuter sous forme de serveurs en utilisant le support du micro-noyau. On a la possibilité d'exécuter ces serveurs de façon privilégiée de façon à ce qu'ils fassent partie intégrante de l'espace d'adressage du micro-noyau. Utiliser un micro-noyau permet donc de faire évoluer librement son système d'exploitation en y ajoutant des serveurs systèmes. Cette notion de modularité est un des atouts majeurs de la technologie micro-noyau.

Les entités essentielles implémentées par le micro-noyau sont les **portes**, les **acteurs** et les **threads**. L'acteur est l'équivalent du processus d'Unix, il définit un espace d'adressage. Un acteur CHORUS peut être composé de plusieurs threads (ou processus légers) qui partagent son espace d'adressage et s'y exécutent de façon concurrente. Le thread est donc l'entité active du micro-noyau. L'intérêt de ces processus légers réside dans le fait qu'il est possible d'exécuter plusieurs traitements de façon concurrente sans avoir à payer le coût de la lourde commutation de contexte habituelle des processus classiques. La porte est le moyen de communication entre deux threads. Le micro-noyau définit des primitives qui permettent d'envoyer un message sur une porte et de recevoir un message sur une porte. L'envoi peut être synchrone ou asynchrone.

Le micro-noyau CHORUS offre également la possibilité de gérer des groupes de portes et différents modes d'envoi de messages sur un groupe de portes.

Le principe même du micro-noyau nous a semblé intéressant pour réaliser notre prototype. En effet, le noyau de sécurité que nous voulons réaliser est une entité finalement simple dont le but est de contrôler de façon relativement automatique les accès à toutes les entités locales. Ainsi, créer un acteur dont le rôle est d'intercepter tous les accès à toutes les portes locales et intégrer cet acteur dans l'espace d'adressage du noyau présente une solution adéquate et simple. De même, la notion de porte comme unique moyen de communication nous permet d'effectuer des contrôles systématiques efficaces sur toutes les communications. Enfin, la notion de threads est particulièrement intéressante pour implémenter un système composé d'objets. En effet, on peut alors implémenter un objet sous la forme d'un acteur dont chacune des méthodes est associée à un processus léger, les méthodes pouvant s'exécuter de façon concurrente.

4.2.2 Le serveur d'autorisation et les noyaux de sécurité

4.2.2.1 Représentation du serveur d'autorisation et des noyaux de sécurité

Dans la configuration utilisée dans le prototype, nous disposons de quatre PC équipés du micro-noyau CHORUS. Ces machines sont connectées au moyen d'un réseau Ethernet. Sur un site s'exécute le serveur d'autorisation et sur chacun des trois autres sites est implémenté un noyau de sécurité.

Ainsi que nous l'avons indiqué, la sécurité du système tout entier repose sur la sécurité du serveur d'autorisation. Ce serveur peut être rendu tolérant aux fautes et aux intrusions par utilisation de la Fragmentation-Redondance-Dissémination. Un prototype d'un tel serveur a déjà été réalisé mais n'est pas à l'heure actuelle porté sur une architecture à micro-noyaux telle que celle que nous utilisons. Aussi, nous avons pour le moment choisi d'implémenter le serveur d'autorisation sous la forme d'un acteur système, sachant que cette implémentation est non sûre mais provisoire.

Chaque noyau de sécurité est implémenté sous la forme d'un acteur système s'intégrant dans le micro-noyau.

Les communications entre serveur d'autorisation et noyaux de sécurité sont

réalisées par échange de messages au travers de portes. Le serveur d'autorisation possède trois portes pour communiquer avec les noyaux de sécurité, chacune lui permettant de communiquer avec un seul des noyaux de sécurité. À chaque porte est associé un thread. Chaque thread est donc en attente de requêtes provenant du noyau de sécurité qui communique avec le serveur d'autorisation via cette porte. Chaque noyau de sécurité quant à lui possède une porte qu'il utilise pour toutes les communications avec le serveur d'autorisation, et deux portes avec lesquelles il va dialoguer avec les autres noyaux de sécurité. Il possède enfin une porte pour dialoguer avec tous les objets locaux. Tous les messages émis par les objets locaux transitent par cette porte et sont donc reçus par le noyau de sécurité. La figure 4.3 représente le schéma de communication entre les noyaux de sécurité et le serveur d'autorisation d'une part et le schéma de communication entre noyaux de sécurité d'autre part.

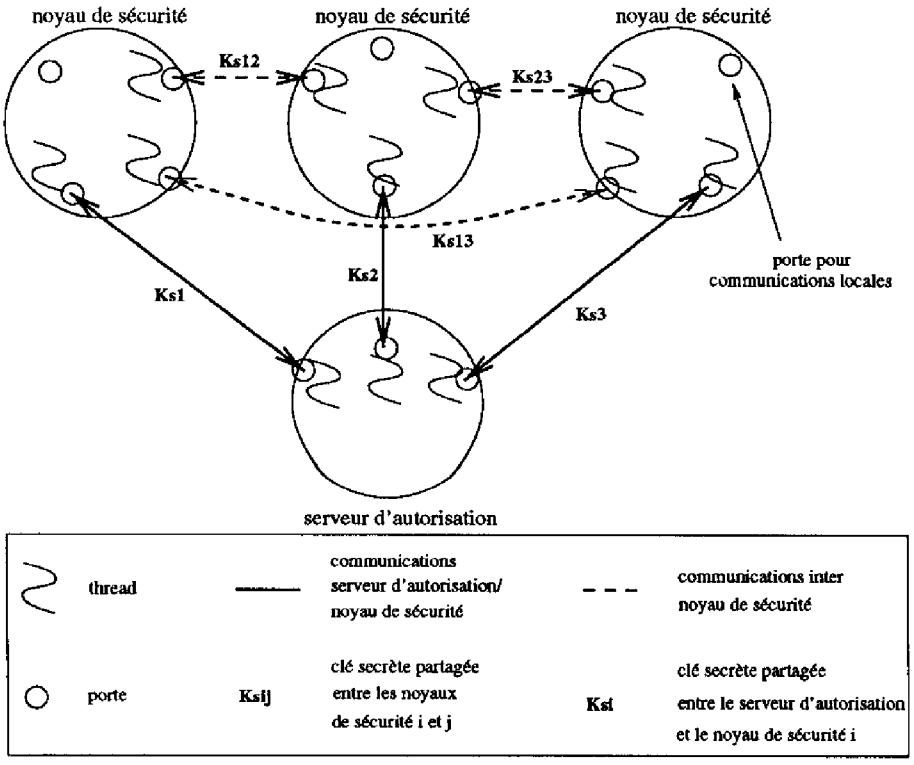


FIG. 4.3 – *Communications entre serveur d'autorisation et noyaux de sécurité*

4.2.2.2 Les communications

Le serveur d'autorisation partage une clé secrète différente avec chacun des noyaux de sécurité. Chaque noyau de sécurité partage une clé secrète différente avec chacun des autres noyaux de sécurité. Ces clés sont échangées lors de l'initialisation du serveur d'autorisation et des noyaux de sécurité. Ces clés sont utilisées pour signer et chiffrer tous les messages qui sont échangés entre les noyaux de sécurité et entre le serveur d'autorisation et les noyaux de sécurité. Il est ainsi possible à chaque noyau de sécurité d'authentifier chaque message qu'il reçoit. Cette propriété est essentielle car les portes grâce auxquelles les entités communiquent sont a priori publiques. L'algorithme de chiffrement utilisé est le DES et les signatures des messages utilisent l'algorithme MD5 [Den83].

4.2.3 La représentation des objets

4.2.3.1 Les informations associées aux objets

Chaque objet s'exécute sous la forme d'un acteur composé de plusieurs threads. Chaque thread représente une méthode de l'objet et est en attente sur une porte associée à cette méthode. À chaque objet qui s'exécute dans le système sont associées les informations suivantes :

- la classe de l'objet ; à chaque classe est associé un programme exécutable qui représente le comportement d'une classe ; plusieurs objets de la même classe seront donc instanciés à partir de ce programme ;
- les portes associées aux différentes méthodes de l'objet ;
- une porte particulière qui est utilisée pour tous les dialogues avec le noyau de sécurité local ; c'est grâce à cette porte que toutes les requêtes émises par un objet vont parvenir au noyau de sécurité qui va pouvoir ainsi identifier l'émetteur de la requête (cf. paragraphe suivant) ;
- une référence unique qui est composée du numéro de site sur laquelle s'exécute l'objet (sur CHORUS, chaque site est identifié par un numéro unique) ainsi que son numéro d'acteur (équivalent à la notion de "pid" Unix).
- le type de l'objet (objet à état ou objet sans état) ;

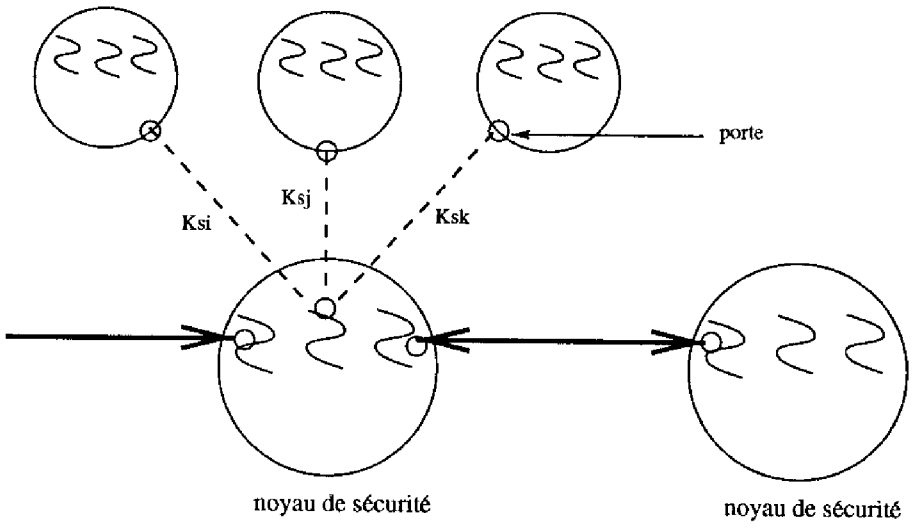
- le type des méthodes (méthodes de lecture, d'écriture ou de lecture-écriture) dans le cas d'un objet à état ;
- le ou les labels de l'objet ;
- l'ensemble des classes que l'objet a le droit d'instancier.

La classe de l'objet, ses labels et les classes qu'il a le droit d'instancier font partie intégrante de l'objet lui-même. Idéalement, ces informations devraient donc être intégrées dans le code de l'objet lui-même. Mais ceci nécessiterait l'intervention d'un compilateur spécifique. Nous avons donc, pour la réalisation de notre prototype, utilisé un fichier que nous avons associé à chaque objet pour décrire ces informations.

Le programme exécutable associé à une classe particulière, ses méthodes ainsi que le mode (lecture-écriture) de ses méthodes sont en revanche des informations qui ne dépendent que de la classe d'un objet et peuvent donc être stockées une fois pour toutes dans le système. Ces informations sont d'ailleurs des informations publiques. Chaque noyau de sécurité en conserve donc une copie ainsi que le serveur d'autorisation (le serveur d'autorisation n'a cependant pas besoin de connaître le programme exécutable associé à une classe puisqu'il n'instancie jamais directement d'objet).

Enfin, les différentes portes associées à un objet ainsi que sa référence sont des données dynamiques qui ne peuvent être connues qu'après l'instanciation de l'objet. Ces données sont donc stockées soit par le serveur d'autorisation, soit par le noyau de sécurité local, soit par les deux à la fois (pour des problèmes d'autorisation certaines données doivent être connues à la fois du serveur d'autorisation et du noyau de sécurité local ; nous revenons sur ce point dans le paragraphe 4.2.4).

À la création de l'objet, celui-ci crée autant de portes que de méthodes de sa classe. La librairie de code que nous avons développée, et qui doit être compilée avec tout objet, se charge de créer une porte pour dialoguer avec le noyau de sécurité local. Les portes créées par l'objet lui-même et qui représentent le moyen d'accès aux méthodes de l'objet (et qui sont donc indispensables à la création de capacités pour ces méthodes) sont transmises au noyau de sécurité local. Dans le cas d'objets persistants, les identificateurs de ces portes sont transmis au serveur d'autorisation (car lui seul peut créer des capacités pour des objets persistants) et ne sont pas conservés par le noyau de sécurité local. Dans le cas d'objets temporaires, le noyau de sécurité local conserve ces identificateurs et les associe à la référence de l'objet.



ksi : clé partagée entre l'objet i et son noyau de sécurité local

FIG. 4.4 – *Communications entre objets et noyau de sécurité*

4.2.3.2 Le cas particulier des fichiers

Nous n'avons pas voulu définir notre propre gestionnaire de fichiers, aussi avons-nous utilisé le gestionnaire de fichiers fourni avec le micro-noyau qui est un gestionnaire Unix. Néanmoins, la gestion Unix des fichiers ne nous satisfait pas car elle ne permet pas de respecter strictement le principe du moindre privilège. Aussi avons-nous décidé d'associer à chaque fichier un acteur et donc, d'effectuer tous les accès à un fichier au travers de l'acteur qui lui est associé. Il nous est possible ainsi de définir des droits respectant le principe du moindre privilège. Chaque acteur représentant un fichier est associé à un programme de la classe FICHIER et à un fichier (au sens Unix). Le programme décrit simplement son comportement et le fichier Unix représente la partie du disque sur laquelle il écrit ou lit ses données.

Il est bien sûr impossible d'avoir un objet actif pour chaque fichier présent dans le système. Aussi avons-nous choisi de ne rendre actif l'acteur représentant un fichier que lorsque celui-ci est concerné par la réalisation d'une opération dans le système. Cette activation s'effectue simplement lorsqu'un objet demande une référence sur un fichier. En effet, chaque objet étant re-

présenté par une référence unique dans le système, tout accès à un objet se fait par l'intermédiaire de cette référence. Ainsi, lorsqu'un objet demande une référence sur un objet qui est un fichier, l'acteur correspondant est activé.

4.2.3.3 Les communications

Nous considérons tout d'abord que localement, les protections mémoires assurent qu'il est impossible à un objet d'aller lire dans un autre espace d'adressage que le sien. Nous utilisons les protections de base offertes par la gestion de la mémoire du micro-noyau.

En revanche, le micro-noyau CHORUS ne protège pas l'utilisation d'une porte particulière : il suffit de connaître le descripteur d'une porte pour émettre depuis cette porte même si on ne l'a pas créée. Ceci nous pose un problème car ceci signifie qu'un objet peut tenter de se faire passer pour un autre objet en essayant d'émettre depuis une porte que lui-même n'a pas créé. La noyau de sécurité n'a alors pas de moyen de savoir réellement qui a émis le message. Aussi avons-nous décidé d'associer à chaque objet une clé secrète lors de sa création. Cette clé est connue du noyau de sécurité (il associe une porte à une clé). Ainsi lorsqu'un objet émet un message, ce message est automatiquement signé en utilisant cette clé secrète. Lorsque le message parvient au noyau de sécurité, il détermine la porte utilisée pour émettre le message. Il génère alors localement une signature du message reçu à l'aide de cette clé. Si la signature reçue et la signature générée coïncident, le serveur de sécurité sait que l'objet qui a émis le message possède bien la clé. Comme nous supposons que l'espace d'adressage d'un acteur ne peut être violé par un autre acteur, le noyau de sécurité peut déduire que le message émis est bien émis par l'acteur associé à cette porte.

4.2.4 Les contrôles discrétionnaires

4.2.4.1 Représentation des capacités

Ainsi que nous l'avons dit, les objets sont représentés par des acteurs composés de plusieurs threads, chaque thread représentant une méthode de l'objet. À chaque thread est associée une porte sur laquelle il reçoit et émet des messages. Dans cette représentation, accéder à une méthode particulière correspond à envoyer un message sur la porte qui lui est associée. Ainsi lorsqu'un objet *o*

est autorisé à invoquer la méthode m' d'un objet o', il reçoit une référence vers la porte qui lui est associée. Une capacité est composée des informations suivantes :

- la référence de l'objet pour laquelle elle est délivrée ;
- la référence de la porte correspondant à la méthode que la capacité permet d'invoquer ;
- le numéro d'activité pour laquelle cette capacité est valide ;
- un numéro identifiant la capacité et permettant d'éviter les rejeux ;
- un champ optionnel indiquant si cette capacité est réutilisable ou pas au cours de l'exécution de la même activité.

Toute capacité transmise du serveur d'autorisation ou d'un noyau de sécurité à un noyau de sécurité est systématiquement chiffrée. Ainsi que nous l'avons expliqué dans le second chapitre, toute capacité doit être uniquement valide pour l'objet pour laquelle elle est générée. Ainsi, nous insérons dans toute capacité la référence de l'objet qui a demandé l'accès correspondant. Toute opération dans le système s'effectue sous la forme d'une activité, identifiable par un numéro d'activité. Chaque capacité comprend donc le numéro de l'activité pendant laquelle elle peut être utilisée. Le champ optionnel permet à un objet d'utiliser plusieurs fois une même capacité lors de l'exécution d'une même activité. En effet, imaginons par exemple, que lors de l'exécution d'une de ses méthodes, un objet accède plusieurs fois à la même méthode d'un objet persistant. Si on ne peut créer de capacité renouvelable, l'objet ne peut alors invoquer plusieurs fois la méthode sans redemander à chaque fois la création d'une nouvelle capacité (à cause du mécanisme de numéro de capacité évitant les rejeux). Si le champ optionnel est positionné à "CAPACITÉ_RÉUTILISABLE", alors l'objet peut réutiliser plusieurs fois la même capacité dans la mesure où tous les accès font partie de la même activité. Soulignons que cette propriété fait qu'il est possible à un objet de rejouer une capacité uniquement pendant la durée d'exécution de l'activité courante.

Les capacités réutilisables permettent de ne pas perdre le coût d'une communication avec le serveur d'autorisation lors de l'accès à un objet. Il faut donc stocker ces capacités. La solution la plus logique semble être de stocker ces capacités dans l'espace d'adressage de l'objet qui en bénéficie. Néanmoins, cette solution pose le problème de la responsabilité de ce stockage. Si l'objet

lui-même stocke cette capacité, cela signifie qu'il est capable de savoir qu'il s'agit d'une capacité réutilisable et il nous semble qu'un objet n'a pas à se préoccuper de cela. Dès lors, nous pensons qu'il préférable que le noyau de sécurité soit responsable de ce stockage et par commodité, nous avons choisi de conserver ces capacités dans l'espace d'adressage du noyau lui-même.

4.2.4.2 Représentation des coupons

Un coupon peut être une capacité ou l'autorisation d'effectuer une opération de haut niveau. Le cas d'une capacité est déjà traité dans le paragraphe précédent. Dans le cas de l'autorisation d'effectuer une opération de haut niveau, le coupon est constitué des informations suivantes :

- la référence de l'objet à qui est accordée cette autorisation ;
- l'opération autorisée ainsi que ses paramètres (ex : **ImprimerFichier(f,p)**).
- le numéro de l'activité pendant l'exécution de laquelle ce coupon est valide ;
- un numéro de coupon (de la même façon que pour les capacités).

Un coupon est valide pour un objet précis qui participe à l'exécution de l'activité en cours. L'identité de l'objet en question doit donc faire partie des informations insérées dans le coupon. Un coupon ne doit pas pouvoir être rejoué (d'où la présence du numéro du coupon) et est utilisable lors de l'exécution d'une activité identifiée dans le coupon.

4.2.4.3 Les informations gérées par le serveur d'autorisation

Il reste enfin à gérer l'ensemble des informations concernant le contrôle d'accès aux objets persistants. Ces informations sont stockées par le serveur de répertoires. Nous avons choisi de fusionner le serveur de répertoires et le serveur d'autorisation en un seul acteur pour des raisons d'efficacité. Nous n'avons pas voulu perdre le coût d'une communication supplémentaire en faisant du serveur d'autorisation et du serveur de répertoires deux acteurs différents. Il est d'ailleurs tout à fait possible de considérer que les données du serveur de répertoires concernant les droits d'accès font partie intégrante du serveur

d'autorisation : ces données sont uniquement consultées par ce dernier. Le serveur de répertoires possède pour chaque objet persistant une entrée décrivant principalement les listes de contrôles d'accès à cet objet. La matrice des droits d'accès est donc stockée sous forme de liste de contrôles d'accès associée à chaque entrée du serveur de répertoire. Pour réaliser ce stockage d'informations, qui est un stockage hiérarchique, nous avons utilisé le gestionnaire de fichiers fourni avec le micro-noyau. Chaque entrée dans le serveur de répertoires est un fichier dont le nom absolu est : */proc/numero_site/pid_obj*[12]. Le contenu typique d'une entrée est représenté ci-dessous.

```
CLASSE FICHIER                          -- la classe de l'objet est FICHIER

PORTES 135634 456456 Lire               -- l'objet possède deux méthodes
       135634 456457 Écrire             -- Lire et Écrire dont les portes
                                        -- d'accès sont : 135634 456456 et
                                        -- 135634 45645

-- LISTES DE CONTRÔLES D'ACCÈS --

BEGIN{ACL}                              -- on définit ici des droits pour des
                                        -- instances, des utilisateurs, des
                                        -- classes ou des rôles

13 4567 :
  livrercourrier this ;                 -- l'objet de référence (13 4567)
                                        -- possède le  droit symbolique
                                        -- ''livrercourrier This''
 toto :
  imprimerfichier this IMPRIMANTE ;     -- toto, créateur du fichier, possède
  écrirefichier this ;                  -- plusieurs droits symboliques sur le
  lirefichier this ;                    -- fichier
  envoyercourrier This rôleA ;

administrateur_courrier :               -- administrateur_courrier est un rôle
  envoyercourrier this utilisateurs;

END{ACL}

-- LISTES DES ATRIBUTS

BEGIN{ATTRIBUTS}
```

12. Nous nous sommes inspirés du répertoire "/proc" du système Solaris2.x qui contient une image de tous les processus en cours d'exécution.

```
FileServer 13 234 ;          -- référence du serveur de fichier est : (13 234)
owner toto ;                 -- créateur du fichier : toto

END{ATTRIBUT}
```

Les règles de création de capacités ne sont en aucun cas liées à un objet particulier : elles expriment des règles générales décrivant les capacités et les coupons qu'un objet reçoit lorsqu'il est autorisé à réaliser une opération élémentaire ou une opération de haut niveau dans le système. Ces règles n'ont par conséquent aucune raison de se trouver dans une entrée du serveur de répertoires correspondant à un objet. Nous avons donc choisi de stocker ces règles dans une entrée particulière du serveur de répertoires (l'entrée *config*). Une règle typique se présente comme indiquée dans la figure 4.6.

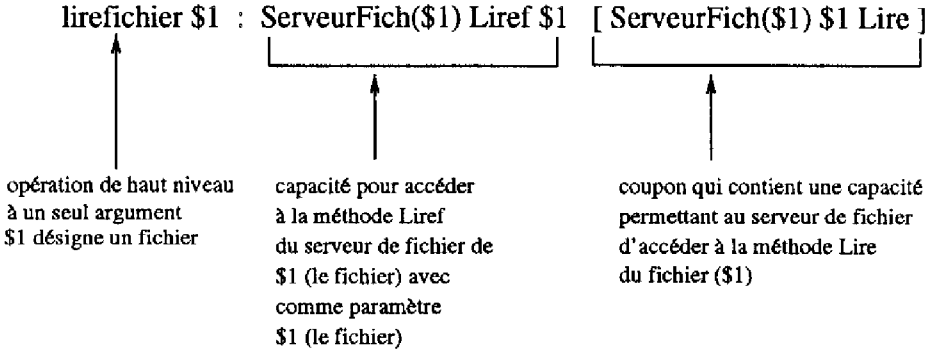


FIG. 4.5 – *Exemple de règle de création de capacités*

4.2.4.4 Scénario de création et vérification des capacités

Invocation

Un objet o demande à invoquer la méthode m d'un objet o'. Après avoir récupéré la référence de l'objet o', l'objet o émet donc une requête du type $Call(ref_{o'}, méthode, paramètres)$. Cette requête est automatiquement reçue par le noyau de sécurité local. Celui-ci vérifie tout d'abord s'il existe une capacité précédemment utilisée (capacité renouvelable) ou reçue mais non utilisée (dans un coupon) lors de l'exécution de cette activité, qui satisfasse l'accès demandé. Si tel est le cas, la capacité est utilisée pour effectuer l'appel. Sinon, le noyau de sécurité transmet l'appel au serveur d'autorisation.

Création de capacité

Le serveur d'autorisation reçoit la requête provenant du noyau de sécurité. Il vérifie tout d'abord si l'accès est autorisé. Il utilise pour cela les listes de contrôle d'accès associées à l'objet cible. Si l'accès est autorisé, le serveur d'autorisation forme alors une capacité (il complète les champs que nous avons décrits dans le paragraphe précédent) et chiffre cette capacité avec la clé secrète qu'il partage avec le noyau de sécurité du site de l'objet cible. Il retourne cette capacité au noyau de sécurité appelant accompagnée de la référence de la porte de l'objet cible correspondante.

Vérification d'une capacité

Lorsqu'un noyau de sécurité reçoit une capacité chiffrée accompagnée d'informations en clair indiquant pour quel accès cette capacité a été créée, le noyau de sécurité déchiffre la capacité avec la clé secrète qu'il partage avec le serveur d'autorisation. Les vérifications effectuées sont les suivantes :

- vérification que les informations contenues dans la capacité correspondent bien à l'accès demandé (vérification de l'identité de l'objet à qui a été donnée la capacité et vérification de la porte à laquelle cette capacité donne accès) ;
- vérification que la capacité n'est pas un rejeu ; soit la capacité correspond à une capacité qui n'a pas encore été reçue, soit elle a déjà été reçue mais le champ optionnel est positionné à "CAPACITÉ_RÉUTILISABLE"et le numéro d'activité contenu dans la capacité correspond bien à l'activité courante.

Si l'ensemble de ces vérifications ne détectent aucune anomalie, alors l'accès est autorisé et l'objet est invoqué.

4.2.4.5 Scénario de création et vérification des coupons

Lorsqu'une règle de création de capacités correspondant à une opération de haut niveau nécessite la création d'un coupon, celui-ci est inséré dans le message destiné au noyau de sécurité et est conservé par le noyau de sécurité. Lorsque le coupon contient l'autorisation d'effectuer une opération de haut niveau, il contient une partie chiffrée par une clé privée du serveur d'auto-

risation. Lorsque le bénéficiaire du coupon va effectuer une demande d'accès correspondant au coupon, deux cas peuvent se produire :

- si le coupon est une capacité, le noyau de sécurité local utilise directement la capacité stockée autorisant l'accès et envoie la demande d'accès au noyau de sécurité concerné ;
- si le coupon contient l'autorisation d'effectuer une opération de haut niveau, le noyau de sécurité l'insère dans le message destiné au serveur d'autorisation ; ce dernier vérifie la validité du coupon (en déchiffrant la partie chiffrée avec sa clé privée) et délivre ensuite la capacité et éventuellement les coupons que lui dicte la règle de création de capacités correspondante.

4.2.5 Les contrôles obligatoires

Le prototype que nous avons élaboré peut fonctionner avec ou sans contrôle obligatoire moyennant très peu de modifications. Ceci est conforme avec la notion d'un noyau de sécurité indépendant de la politique de sécurité que nous avons voulu élaborer. Ainsi, l'ajout des contrôles obligatoires dans notre prototype a donné lieu à un travail relativement simple. Il a consisté simplement pour chaque accès à comparer les labels du message et ceux de l'objet en fonction des règles du schéma d'autorisation.

4.2.5.1 Représentation des labels

Les objets

Les caractéristiques d'un objet utilisées par les contrôles obligatoires sont ses labels et le mode de ses méthodes (méthode de lecture, d'écriture, de lecture-écriture). Le mode de ses méthodes est en fait uniquement lié à la classe de l'objet. Ces informations sont donc stockées avec le descriptif de la classe elle-même. Les labels d'un objet sont contenus dans un fichier associé au code de l'objet. Comme il ne doit pas être possible de modifier ces labels, il est clair que cette solution pose des problèmes. Aussi, les labels sont chiffrés par une clé secrète détenue par le noyau de sécurité qui a créé l'objet. Seul, le noyau de sécurité peut donc consulter ces labels lorsqu'une activité veut accéder à un objet. Notons que cette solution, si elle est satisfaisante du point de vue

de la confidentialité, rend plus difficile la migration d'objets persistants dans le système, propriété qui est implémentée dans le micro-noyau CHORUS.

Les requêtes

Pour être fidèle à notre modèle d'autorisation, il est nécessaire d'étiqueter toute activité du système et donc toute requête émise au sein du système. Pour cela, à chaque message transmis est associé l'ensemble d'informations suivantes : l'identification de l'utilisateur ayant déclenché l'exécution de l'activité, le numéro de l'activité dont le message fait partie et deux labels.

L'implémentation de cette structure a été particulièrement aisée car l'interface du micro-noyau CHORUS permet d'associer à chaque message, un autre petit message de taille fixe (appelé *annexe* du message). Nous avons choisi d'utiliser cette annexe pour y insérer les informations ci-dessus. L'avantage de cette implémentation est que si l'on désire ne pas effectuer de contrôles obligatoires, le noyau de sécurité peut alors tout simplement ignorer l'annexe. Ceci aurait été plus compliqué si les informations contenues dans l'annexe avaient été contenues dans le message lui-même.

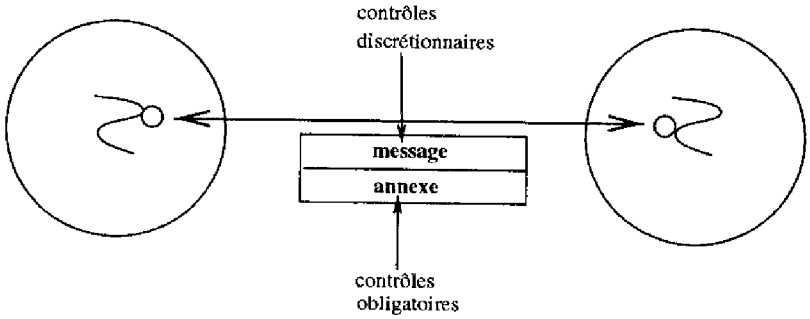


FIG. 4.6 – *Envoi d'un message accompagné de son annexe*

4.2.5.2 Création et vérification des labels

Pour certains objets tels que les serveurs et les démons, les labels sont fixés une fois pour toutes lors de l'évaluation de leurs codes et donc insérés dans ce code. En revanche, pour les objets de type fichier, qui sont destinés à recevoir une information, le label qui leur est affecté dépend de la requête qui les crée et des règles de la politique obligatoire. C'est pourquoi le label d'un fichier est créé dynamiquement lors du premier accès en écriture à cet objet, et il est

conservé chiffré par le noyau de sécurité local.

Lorsqu'une requête demande à accéder à un objet local, le noyau de sécurité extrait de l'annexe accompagnant tout message les labels de l'activité courante et les compare au label de l'objet conformément aux règles du schéma d'autorisation. Pour cela, il détermine, en fonction d'informations locales, la classe de l'objet invoqué et par conséquent la famille de l'objet (objet à état ou objet sans état) et le mode de la méthode invoquée (lecture, écriture ou lecture-écriture).

Les contrôles obligatoires s'effectuent systématiquement lorsque des contrôles discrétionnaires s'effectuent. De plus, tous les retours de fonctions (dans le cas d'un objet à état seulement) doivent être contrôlés également pour empêcher un éventuel flux d'informations. En effet, il peut y avoir un échange d'information au retour de l'exécution d'une fonction (exécution d'une méthode de lecture notammment).

4.3 Conclusion

Ainsi que nous l'avons dit cette implémentation est un prototype. En ce sens, elle ne présente pas les garanties que présenterait la solution consistant à modifier directement le micro-noyau CHORUS en ajoutant des champs supplémentaires dans les structures décrivant les messages, les acteurs et en ajoutant des contrôles sur tous les échanges de messages. Le prototype vient en fait démontrer que ce type de projet a un sens et est efficace. Il est bien sûr contournable dans la mesure où tous les objets introduits dans le système doivent être compilés avec une librairie permettant la communication avec le noyau de sécurité. Les communications "classiques", sans contrôle, ne sont pas directement interdites. Néanmoins, cette implémentation permet de montrer que la mise en œuvre des principes que nous avons élaborés est réalisable et que l'utilisation d'un micro-noyau tel que CHORUS rend cette mise en œuvre relativement aisée.

Conclusion générale

Bilan

Dans ce mémoire, nous avons apporté de nouvelles idées quant à la protection des systèmes répartis et plus particulièrement des systèmes répartis modélisés sous la forme d'objets. Notre démarche a été la suivante :

- Nous avons tout d'abord analysé la vision classique de la protection et nous avons dégagé les principes qui nous semblent fondamentaux pour la protection de systèmes répartis : éviter l'approche centralisée qui est trop coûteuse en performances et dont la sécurité repose sur un seul site, éviter l'approche du livre rouge qui repose sur un excès de confiance entre les différentes TCB. Notre architecture est ainsi composée d'un serveur d'autorisation centralisé (que l'on peut rendre tolérant aux fautes et aux intrusions en utilisant la Fragmentation-Redondance-Dissémination) et de noyaux de sécurité sur chaque site du système. Chaque noyau de sécurité ne fait confiance qu'au seul serveur d'autorisation.
- Nous avons élaboré un schéma d'autorisation qui répond le mieux possible aux besoins que nous nous sommes fixés : gestion de l'exécution d'opérations complexes dans le système, respect du principe du moindre privilège, facilité d'administration. Nous avons ainsi défini de nouveaux droits d'accès (appelés droits symboliques) et un schéma de gestion de ces droits au niveau du serveur d'autorisation. Nous avons également présenté un nouveau schéma de délégation de droits par des coupons.
- Nous avons ensuite montré que le schéma d'autorisation que nous avons élaboré peut être utilisé dans le cadre d'une politique obligatoire multiniveau. Nous avons pour cela défini un nouveau modèle de politique de sécurité multiniveau adapté aux concepts de la programmation par objets.

- Nous avons finalement décrit une implémentation de ce schéma d'autorisation réalisée sur une architecture répartie composée de machines à base de micro-noyaux.

Perspectives

Il est tout d'abord nécessaire d'étudier précisément la collaboration entre le serveur de sécurité développé dans le cadre du projet Delta-4 et les différents noyaux de sécurité que nous avons développés. Ce travail n'est pas immédiat car le serveur d'authentification et d'autorisation tolérant les fautes et les intrusions est composé de plusieurs sites de sécurité. La gestion des communications entre serveur d'autorisation et noyaux de sécurité est donc un travail conséquent : chaque requête provenant d'un noyau de sécurité doit être diffusée à l'ensemble des sites de sécurité. Cette interconnexion serait à étudier en détail.

Il serait également intéressant d'implémenter directement un micro-noyau de sécurité en modifiant le cœur même du micro-noyau CHORUS et en y intégrant des contrôles d'accès. Notre travail a montré que cette réalisation est possible sans être extrêmement coûteuse. Nous avons déterminé assez précisément les contrôles à réaliser. Le prototype donne quelques éléments de réponse pour faciliter cette réalisation. Il serait à présent intéressant de les intégrer directement au sein du micro-noyau de façon à obtenir une version sécurisée du micro-noyau CHORUS susceptible d'être inclus dans un système composé d'objets répartis.

Cependant, notre travail n'a fait qu'aborder le problème de l'utilisation de notre schéma d'autorisation dans le cadre de réseaux à grande échelle. Il reste du travail à effectuer dans ce cadre, notamment en ce qui concerne l'interconnexion des différents serveurs d'autorisation, la façon dont on peut gérer la cohérence des informations qu'ils détiennent, la façon dont il faut protéger ces informations et surtout toutes les communications qui permettent d'échanger ces informations. Notre travail donne quelques éléments de réponse mais il serait intéressant d'aller plus loin et de réaliser de la même façon un prototype composé de plusieurs domaines gérés par des serveurs d'autorisation différents.

Il est intéressant enfin d'étudier ce modèle dans le cadre de politiques de sécurité visant à protéger l'intégrité des informations. De nombreux problèmes sont à résoudre : si l'on imagine aisément que l'on peut conserver la notion d'activités dans un tel système, la notion d'objet à état et d'objets sans état

a-t-elle un sens? comment attribuer des labels d'intégrité aux différentes entités? Autant de questions et de développements ouverts qui justifient la poursuite de ces travaux dans le but de faire progresser la protection des systèmes informatiques de l'avenir.

Annexe A

Glossaire

Nous regroupons ici les principales expressions utilisées dans les chapitres 2, 3 et 4.

$R\hat{o}le(u)$: ensemble des rôles de l'utilisateurs u.

$SousClasse(C)$: ensemble des classes composé de la classe C et de ses descendantes.

$\mathbf{op(arg_1, ..., arg_n)}$: opération de haut niveau **op** appliquée aux objets $arg_1, ..., arg_n$.

$\mathbf{A(op)(e, arg_1, ..., arg_n)}$: autorisation d'exécuter l'opération de haut niveau $\mathbf{op(arg_1, ..., arg_n)}$ pour l'entité e (e pouvant être un utilisateur, un rôle, un objet ou une classe).

$\mathbf{ds(arg_1, ..., arg_n)}$: droit symbolique s'appliquant aux objets $arg_1, ..., arg_n$.

$\mathbf{ds(arg_1, ..., arg_{i-1}, this, arg_{i+1}, ..., arg_n)}$: représentation d'un droit symbolique placée dans la colonne de l'argument arg_i dans la matrice d'accès.

$$RS(e, op, ds, arg_1, ..., arg_n) : \frac{\mathbf{ds_{e,arg_1}(arg_1, ..., arg_n)}, ..., \mathbf{ds_{e,arg_n}(arg_1, ..., arg_n)}}{\mathbf{A(op)(e, arg_1, ..., arg_n)}}$$

Règle de droits symboliques : elle représente le fait que si l'entité e possède dans la matrice d'accès, pour chacun des arguments arg_i un droit symbolique

dont les autres arguments constituent un sur-ensemble de
$arg_1, ..., arg_{i-1}, arg_{i+1}, ..., arg_n$, alors cette entité est autorisée à exécuter l'opération de haut niveau $\mathbf{op(arg_1, ..., arg_n)}$.

Un **niveau de sécurité** est défini par une classification (ou une habilitation dans le cas d'un utilisateur) et un compartiment. Un compartiment est un ensemble de catégories, une catégorie représentant le domaine de l'information. Les niveaux de sécurité sont ordonnés selon l'ordre partiel suivant : le niveau de sécurité d'une information I défini par sa classification c et son compartiment C est dominée par le niveau de sécurité d'une autre information I' défini par sa classification c' et son compartiment C' si et seulement si $c \leq c'$ et $C \subseteq C'$. Réciproquement, on dira que le niveau de I' domine celui de I si et seulement si les mêmes conditions sont remplies. Nous notons cette relation de dominance : $\preceq$.

Si A et B sont deux niveaux de sécurité, $A \prec B$ est équivalent à :
$(A \preceq B)$ et (A *différent de* B).

Si A et B sont deux niveaux de sécurité, l'intervalle $[A, B]$ est ainsi défini :
$\forall x \in [A, B] \Leftrightarrow A \preceq x \preceq B$.

La relation $A \not\preceq B$ est équivalente à :
$(B \prec A)$ ou (A *et* B *sont non comparables*).
A et B sont non comparables si à la fois le compartiment de A n'est pas inclus dans celui de B et le compartiment de B n'est pas inclus dans celui de A.

Annexe B

Opération de haut niveau : écrire un fichier

Cette annexe présente la matrice des droits d'accès ainsi que les différentes règles (règles de création de capacités et règles de droits symboliques) utilisées dans notre prototype pour réaliser l'opération de haut niveau : "l'utilisateur u écrit des données dans le fichier f". Le serveur de fichier qui intervient dans la réalisation de cette opération est l'objet sf.

B.1 La matrice des droits d'accès

	f	...
u	`EF(this)`	...
sf	Read	...
...		

Définition des droits symboliques :

`EF(f)` : $(Classe(f) \in SousClasse(FICHIER))$

B.2 Les règles de droits symboliques

La règle suivante doit être incluse dans l'ensemble des règles de droits symboliques :

$$\text{(RS1)}\quad RS(e, écrirefichier, EF, f) : \frac{\mathtt{IF_f(f)}}{\mathbf{A(écrirefichier)(e, f)}}$$

Cette règle signifie que pour être autorisé à exécuter l'opération "modifier le fichier f", une entité e doit posséder dans la matrice des droits d'accès le droit symbolique : `EF` pour le fichier f.

B.3 Les règles de création de capacités

L'ensemble de ces règles doit au moins contenir la règle suivante :

(RC1) $\forall f$ *instance de* $FICHIER$,
$Cap(\mathbf{o}, \mathbf{écrirefichier(f)}) ::=$
$\{Cap(o, \mathsf{ServeurFich}(f).Écrivef(f)),\ [\mathbf{Cap}(\mathsf{ServeurFich}(\mathbf{f}), \mathbf{f.Écrire}())]\}$

Notons que dans cette règle, nous insérons le coupon permettant au serveur de fichier de f d'invoquer la méthode *Écrire* du fichier f. Ce coupon n'est pas nécessaire dans la mesure où ce droit fait partie de la matrice des droits d'accès mais il permet un gain en performance. En effet, lorsque fs invoque la méthode *Écrire* de f, le noyau de sécurité local qui a conservé le coupon lors de l'accès à la méthode *Écrivef* de sf, identifie ce coupon comme valide pour l'accès et l'insère dans la requête. Ainsi, on évite un passage par le serveur d'autorisation.

B.4 Le scénario

- L'utilisateur u demande à réaliser l'opération de haut niveau **Écrirefichier(f)**. La règle de droits symboliques SR1 autorise cet accès puisque l'utilisateur u possède dans la matrice le droit symbolique `EF(this)` pour le fichier f. La règle de création de capacités RC1 indique au serveur d'autorisation de fournir à l'analyseur de commandes s'exécutant

pour le compte de l'utilisateur u, la capacité permettant d'invoquer la méthode *Écrire*f de sf et le coupon permettant à sf d'invoquer la méthode *Écrire* de f.

- L'utilisateur invoque la méthode *Écrire*f de sf et présente la capacité reçue du serveur d'autorisation. La validité de cette capacité est vérifiée par le noyau de sécurité local. Le coupon qui accompagne la requête est transmis à sf.

- Le serveur de fichiers sf invoque la méthode *Écrire* de f. Le noyau de sécurité local autorise cet accès grâce au coupon qui a été généré pour pour sf.

Annexe C

Opération de haut niveau : lire son courrier électronique

Cette annexe présente la matrice des droits d'accès ainsi que les différentes règles (règles de création de capacités et règles de droits symboliques) utilisées dans notre prototype pour réaliser l'opération de haut niveau : "l'utilisateur u consulte son courrier électronique". Les objets qui interviennent dans la réalisation de cette opération de haut niveau sont les différents messages de la boîte aux lettres de u (chaque message est représenté par une instance de la classe FICHIER_MAIL, qui est une sous-classe de la classe FICHIER), le serveur de fichier sf de classe SERVFICH, l'objet mua de classe MUA (tiré de l'anglais "Mail User Agent") qui fournit à l'utilisateur u une interface graphique lui permettant de visualiser ses messages.

C.1 La matrice des droits d'accès

	u	message1	message2	...
u	LC(this)			...
mua		LF(this)	LF(this)	...
...				

Définition des droits symboliques :

LC(ut) : (*ut est un utilisateur*));
LF(f) : ($Classe(f) \in SousClasse(FICHIER)$);

C.2 Les règles de droits symboliques

La règle suivante doit être inclue dans l'ensemble des règles de droits symboliques :

$$\text{(RS1)}\quad RS(e, lirecourrier, LC, ut) : \frac{\mathtt{LC_{ut}(ut)}}{\mathbf{A(lirecourrier)(e, ut)}}$$

$$\text{(RS2)}\quad RS(e, lirefichier, LF, f) : \frac{\mathtt{LF_f(f)}}{\mathbf{A(lirefichier)(e, f)}}$$

La règle RS1 signifie que pour être autorisée à exécuter l'opération "lire le courrier d'un utilisateur *ut*", une entité *e* doit posséder dans la matrice des droits d'accès le droit symbolique LC pour l'utilisateur *ut*. La règle RS2 signifie que pour être autorisée à exécuter l'opération de haut niveau "lire le fichier *f*", une entité *e* doit posséder dans la matrice des droits d'accès le droit symbolique LF pour le fichier *f*.

C.3 Les règles de création de capacités

L'ensemble de ces règles doit au moins contenir les règles suivantes :

(RC1) $\forall ut$ *utilisateur*,
$Cap(\mathbf{o}, \mathbf{lirecourrier(ut)}) ::= Cap(o, !MUA.showmail(ut))$
(RC2) $\forall f$ *instance de* $FICHIER$,
$Cap(\mathbf{o}, \mathbf{lirefichier(f)}) ::=$
$\{Cap(o, \mathsf{ServeurFich}(f).Liref(f)), [\mathbf{Cap}(\mathsf{ServeurFich}(\mathbf{f}), \mathbf{f.Lire}())]\}$

Notons que lorsqu'une entité est autorisée à exécuter l'opération **lirecourrier(ut)** (RC1), elle reçoit la capacité lui permettant d'accéder à la méthode *showmail* de l'instance locale de la classe MUA mais aucun coupon. Il aurait pourtant été intéressant de délivrer des coupons autorisant pour l'instance

locale de la classe MUA la lecture de chacun des messages de la boîte aux lettres de *ut*. Nous n'avons ceppendant pu réaliser cette création de coupons car le pouvoir d'expression de nos règles ne nous permet pas d'exprimer le nombre de messages d'une boîte aux lettres. La règle RC2 signifie que lorsqu'une entité est autorisée à réaliser l'opération **lirefichier(f)**, elle reçoit une capacité lui permettant d'accéder à la méthode *Lire*f du serveur de fichiers de f accompagnée du coupon permettant au serveur de fichiers de f d'accéder à la méthode *Lire* du fichier f.

C.4 Le scénario

- L'utilisateur u demande à consulter son courrier par l'intermédiaire de l'opération de haut niveau **lirecourrier(u)**. La règle de droits symboliques RS1 autorise cet accès puisque l'utilisateur u possède le droit symbolique `LC(this)` dans la colonne u de la matrice des droits d'accès. Ainsi que le dicte la règle RC1, la capacité permettant à u (c'est-à-dire à l'objet *shell* s'exécutant pour u) d'accéder à la méthode *show_mail* de *mua*.

- L'utilisateur u invoque la méthode *show_mail* de *mua*. Le noyau de sécurité local vérifie que cette requête est accompagnée d'une capacité autorisant l'accès.

- L'objet *mua* consulte la liste des messages pour l'utilisateur u et demande pour chacun d'eux à réaliser l'opération de haut niveau : **lirefichier(message)**. La règle (RS2) et la matrice de droits d'accès autorisent cette opération. Conformément à la règle de création de capacités, *mua* reçoit la capacité d'accéder à la méthode *Lire*f de sf ainsi qu'un coupon pour sf qui lui permet d'accéder à la méthode *Lire* du message en question.

- L'objet *mua* accède à la méthode *Lire*f de sf (accès contrôlé par le noyau de sécurité local) et transmet à sf le coupon lui permettant d'accéder au message courant.

- Le serveur de fichier sf utilise le coupon reçu de *mua* pour accéder à la méthode *Lire* du message.

- Les contenus des messages reviennent à *mua* sous la forme de deux messages de retour.

- L'objet *mua* affiche les contenus des messages à l'utilisateur u.

Bibliographie

[ABC+96] J. Arlat, J.P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, J.C. Laprie, C. Mazet, D. Powell, C. Rabéjac, and P. Thévenod. *Guide de la Sûreté de Fonctionnement.* Cépaduès, 2ème édition, 1996.

[ADG+92] R. Ahad, J. Davis, S. Gower, P. Lyngbaek, A. Marynowski, and E. Onuegbe. Supporting Access Control in an Object-Oriented Database Language. In *Proc. of the Third International Conference on Extending Database Technology (EDBT)*, pages 184–220, Vienne (Autriche), 1992. Springer-Verlag LNCS.

[BCCGY93a] N. Boulahia-Cuppens, F. Cuppens, A. Gabillon, and K. Yazdanian. Multilevel Security in Object-Oriented Databases. In B. Thuraisingham, R. Sandhu, and T.C. Ting, editors, *Proc. of the OOPSLA 93 Conference Worshop on Security in Object-Oriented Systems*, pages 79–89, Washington DC, septembre 1993. Springer-Verlag.

[BCCGY93b] N. Boulahia-Cuppens, F. Cuppens, A. Gabillon, and K. Yazdanian. Multiview Model for Object-Oriented Databases. In *Proc. of the ninth Annual Computer Security Applications Conference*, Orlando, Florida, décembre 1993.

[BD90] L. Blain and Y. Deswarte. Intrusion-Tolerant Security Servers for delta-4. In *Proc. of the ESPRIT'90 Conference*, pages 355–370, Bruxelles, Belgique, CEC-DGXIII, novembre 1990. Kluver Academic Publishers. ISBN 0-7923-1039-X.

[Bel88] D.E. Bell. Concerning Modeling of Computer Security. In *Proc. of the 1988 IEEE Symposium on Security and Privacy*, pages 8–13, Oakland, Californie, mai 1988.

[Ber92] Elisa Bertino. Data Hiding and Security in an Object-Oriented Database System. In *Proc. of the Eighth IEEE International Conference on Data Engineering*, pages 338–347, Phoenix, Arizona, février 1992.

[Ber96] Y. Bergeon. *Les Canaux Cachés dans les Protocoles Réseaux.* PhD thesis, Université Paris XI - Institut d'Electronique Fondamentale, 1996.

[Bib77] K.J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR 76-372, MITRE Co., avril 1977.

[BL74] D.E. Bell and L.J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report M74-244, MITRE Co., octobre 1974.

[BL75] D.E. Bell and L.J. LaPadula. Secure Computer Systems: unified Exposition and Multics Interpretation. Technical Report MTR-2997, MITRE Co., juillet 1975.

[Bla92] L. Blain. *La Tolérance aux Fautes pour la Gestion de la Sécurité dans les Systèmes Répartis.* PhD thesis, Institut National Polytechnique de Toulouse, 1992.

[BN89] F.C. Brewer and D.M.J. Nash. The Chinese Wall Security Policy. In *Proc. of the 1989 Symposium on Research in Security and Privacy, IEEE society Press*, pages 206–214, Oakland, Californie, mai 1989.

[BPB+90] M.A. Branstad, C.P. Pfleeger, D. Brewer, C. Jahl, and H. Kurth. Apparent Differences Between the US TCSEC and the European ITSEC. Trusted Information Systems, 1990.

[Bry94] C. Bryce. *Etude et Mise en Oeuvre de Propriétés dans les Systèmes Informatiques.* PhD thesis, Institut de Formation Supérieure en Informatique et Communication, 1994.

[BTM88] M. Branstad, H. Tajalli, and F. Mayer. Security Issues of the Trusted MACH System. In *Proc. of Fourth Aerospace Computer Security Applications Conference*, pages 362–367, Orlando, Florida, décembre 1988.

[Cal95] C. Calas. *Pour une Protection Efficace des Données et des Traitements dans les Systèmes Informatiques Répartis.* PhD thesis, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, 1995.

[CC96] Common Criteria for Information Technology Security Evaluation, Part1: Introduction and General Model. CCEB-96/011, janvier 1996.

[CHO93] Chorus/Fusion for SCO, Programmer's Reference Manual. Technical Report CS/TR-93-86, Chorus Systems, 1993.

[CHO94] Chorus/Fusion for SCO, Programmer's Guide. Technical Report CS/TR-93-77.2, Chorus Systems, 1994.

[CLS92] L-F. Cabrera, A.W. Luniewski, and J.W. Stamos. Fine-Grained Access Control in a Transactional Oobject-Oriented System. *Computing Systems*, 5(3):199–216, 1992.

[CTC93] The Canadian Trusted Computer Product Evaluation Criteria. version 3.0e, Canadian System Security Center, Communications Security Establishment of Canada, janvier 1993.

[CW87] D.D. Clark and D.R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proc. of the 1987 Symposium on Research in Security and Privacy, IEEE society Press*, pages 184–194, Oakland, Californie, mai 1987.

[d'A94] B. d'Ausbourg. Implementing Secure Dependencies Over a Network by Designing a Distributed Security Subsystem. In *Proc. of European Symposium on Research in Computer Security*, pages 249–266, Brighton (UK), novembre 1994.

[Dac94] M. Dacier. *Vers une Évaluation Quantitative de la Sécurité Informatique.* PhD thesis, Institut National Polytechnique de Toulouse, 1994.

[DBF91] Y. Deswarte, L. Blain, and J.C. Fabre. Intrusion Tolerance in Distributed Computing Systems. In *Proc. of Symposium on Research in Security and Privacy, IEEE society Press*, pages 110–121, Oakland, Californie, mai 1991.

[DBWL95] R.H. Deng, S.K. Bhonsle, W. Wang, and A. Lazar. Integrating security in CORBA Based Object Architectures. In *Proc. of Symposium on Research in Security and Privacy, IEEE society Press*, pages 50–61, Oakland, Californie, mai 1995.

[Den83] D. Denning. *Cryptography and Data Security.* Addison-Wesley, ISBN 0-201-10150-5, 1983.

[Des91] Y. Deswarte. *Construction des Systèmes d'Exploitation Répartis*, chapter Tolérance aux Fautes, Sécurité et Protection, pages 1–50. Collection Didactique, INRIA, 1991.

[DoD85] U.S. Departement of Defense Trusted Computer Security Evaluation Criteria (TCSEC). 5200.28-STD, décembre 1985.

[Fab74] R.S. Fabry. Capability-Based Addresing. *Communications of the ACM*, pages 403–412, juillet 1974.

[FC92] Federal Criteria for Information Technology Security. Draft, Volume I et II, National Institute of Standards and Technology (NIST) and National Security Agency (NSA), 1992.

[Fer90] D. Ferbrache. *The Pathology of Computer Viruses.* Springer-Verlag, ISBN 3-450-19610-2, 1990.

[FNP+95] JC. Fabre, V. Nicomette, T. Pérennou, R. Stroud, and Z. Wu. Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming. In *Proc. of 25th Int. Symp. on Fault Tolerant Computing (FTCS-25)*, pages 489–498, Pasadena, Californie, juin 1995.

[FP85] J.D.S. Fraga and D. Powell. A Fault and Intrusion-Tolerant File System. In *Proc. of 3rd International Congress on Computer Security (IFIP/SEC'85)*, pages 203–218, Dublin, Irlande, août 1985.

[Fra83] L.J. Fraim. Scomp: A Solution to the Multilevel Security Problem. *IEEE Computer*, 16(7):26–34, juillet 1983.

[Gas88] M. Gasser. *Building a Secure Computer System.* Van Nostrand Reinhold Comany, New York, 1988.

[GM84] J. Goguen and J. Meseguer. Unwinding and Inference Control. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, pages 75–86, Oakland, Californie, mai 1984.

[Gon89] Li Gong. A Secure Identity-Based Capablity Systems. In *Proc. of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, Oakland, Californie, mai 1989.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, 1983.

[Hag93] D. Hagimont. *Adressage et Protection dans un Système Réparti.* PhD thesis, Institut National Polytechnique de Grenoble, 1993.

[HRU76] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, août 1976.

[Hu91] W-M. Hu. Reducing Timing Channels with Fuzzy Time. In *Proc. of the 1991 IEEE Symposium on Security and Privacy*, pages 8–20, Oakland, Californie, mai 1991.

[IP81] Internet Protocol. RFC 791, septembre 1981.

[ITS91] Information Technology Security Evaluation Criteria. European Communities, juin 1991.

[ITU93] The Directory: Overview of Concepts Models and Services. ITU Recommendation X500, novembre 1993.

[JCS92] The Japanese Computer Security Evaluation Criteria - Functionality Requirements. Draft V1.0. Ministry of International Trade and Industry, août 1992.

[JK90] S. Jajodia and B. Kogan. Integrating an Object-Oriented Data Model with Multi-Level Security. In *Proc. of the 1990 IEEE Symposium on Security and Privacy*, pages 48–69, Oakland, Californie, mai 1990.

[Kem82] R.A. Kemmerer. A Practical Approach to Identifying Storage and Timing Channel. In *Proc. of the 1982 IEEE Symposium on Security and Privacy*, pages 66–73, Oakland, Californie, 1982.

[KN93] J. Kohl and C. Neuman. The Kerberos Network Authentification Service (V5). *RFC 1510*, septembre 1993.

[Kra80] S. Krakowiak. *Principes des Systèmes d'Exploitation des Ordinateurs*, chapter Liaison et Désignation des Objets, pages 1–50. DUNOD Informatique, 1980.

[KTT89] T.F. Keefe, W.T. Tsai, and M.B. Thuraisingham. SODA: a Secure Object-oriented Database System. *Computers and Security*, 8(6):517–533, 1989.

[Lam73] B.W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, octobre 1973.

[Lam74] B.W. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, 1974.

[Lan81] C.E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 3(13):247–278, septembre 1981.

[Lan83] C.E. Landwehr. The Best Available Technologies for Computer Security. *IEEE Computer*, 16(7):86–100, juillet 1983.

[LH89] E.E.O. Roos Lindgreen and I.S. Herschberg. On the Validity of the Bell-LaPadula Model. *Computers and Security*, 23(2):39–44, avril 1989.

[Loe94] K. Loepere. The Covert Channel Limiter Revisited. *ACM SIGOPS Operating Systems Review*, 23:317–333, 1994.

[LS77] R.J. Lipton and L. Snyder. A Linear Time Algorithm for Deciding Subject Security. *Journal of the ACM*, 24(3):455–464, juillet 1977.

[LSC91] A.W. Luniewski, J.W. Stamos, and L-F. Cabrera. A Design for Fine-Grained Access Control in Melampus. In *Proc. of the 1991 IEEE International Workshop on Object Orientation in Operating Systems (IWOOPS)*, pages 185–189, Palo Alto, Californie, Octobre 1991.

[Lun89] T.F. Lunt. Multilevel Security for Object-Oriented Database Systems. In D.L. Spooner and C.E. Landwher, editors, *Proc. IFIP WG 11.3 Workshop on Database Security*, pages 199–209, Monterey, Californie, septembre 1989. North-Holland.

[McL85] J. McLean. A Comment on The Basic Security Theorem of Bell and LaPadula. Information Processing Letters, 20, 1985.

[McL87] J. McLean. Reasoning About Security Models. In *Proc. of the 1987 IEEE Symposium on Security and Privacy*, pages 123–130, Oakland, Californie, mai 1987.

[Mey88a] B. Meyer. *Conception et Programmation par Objet.* Prentice Hall, 1988.

[Mey88b] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Min78] N. Minsky. *Foundations of Secure Computation*, chapter The Principle of Attenuation of Privileges and its Ramifications, pages 255–276. Academic Press, 1978.

[ML92] J.K. Millen and T.F Lunt. Security for Object-Oriented Database Systems. In *Proc. of the 1992 IEEE Symposium on Security and Privacy*, pages 260–272, Oakland, Californie, mai 1992.

[NCS87] A Guide to Understanding Discretionnary Access Control in Trusted Systems. National Computer Security Center, septembre 1987.

[ND94] V. Nicomette and Y. Deswarte. An Authorization Scheme for Distributed Object Systems. In *Proc. of the ACM OOPSLA'94 Conference Workshop on standards for security in Object-Oriented Systems*, pages 1–9, Portland, Oregon, octobre 1994.

[ND96a] V. Nicomette and Y. Deswarte. A Multilevel Security Model for Distributed Object Systems. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, pages 80–98, Rome, Italie, septembre 1996.

[ND96b] V. Nicomette and Y. Deswarte. Symbolic Rights and Vouchers for Access Control in Distributed Object Systems. In *Proc. of ASIAN 96 Conference*, pages 192–203, Singapour, décembre 1996.

[Nic95] V. Nicomette. Un Schéma d'Autorisation pour la Protection d'Objets Répartis. In *Deuxièmes Journées Jeunes Chercheurs en Systèmes Répartis*, page 8p., Rennes, octobre 1995.

[NS78] R.M. Needham and M.D. Schroeder. Using Encryption for Authentification in Large Networks in Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[OMG94a] Object Services RFP 3. OMG TC Document 94-7-1, juillet 1994.

[OMG94b] OMG White Paper on Security. OMG Security Working Group, avril 1994.

[OMG95] Corba Security. OMG TC Document 95-12-1, décembre 1995.

[Par91] T.A. Parker. A Secure European System for Applications in a Multi-vendor Environment (The SESAME Project). In *Proc. of the 14th National Computer Security Conference, NCSC and NIST*, pages 505–513, Washington, octobre 1991.

[PK91] P.A. Porras and R.A. Kemmerer. Covert Flow Trees: A Technique for Identifying and Analyzing Covert Storage Channels. In *Proc. of the 1991 IEEE Symposium on Security and Privacy*, pages 36–51, Oakland, Californie, 1991.

[Ray87] M. Raynal. *Systèmes Répartis et Réseaux: concepts, outils et algorithmes.* Eyrolles, Paris, 1987.

[RR83] J. Rushby and B. Randell. A Distributed Secure System. *IEEE Computer*, 16(7):55–67, juillet 1983.

[RSC92] J. Richardson, P. Schwarz, and L-F. Cabrera. CACL: Efficient Fined-Grained Protection for Objects. In B. Thuraisingham, R. Sandhu, and T.C. Ting, editors, *Proc. of the ACM Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 263–275, Vancouver (Canada), octobre 1992. Springer-Verlag.

[San92] R.S. Sandhu. The Typed Access Matrix Model. In *Proc. of the 1992 IEEE Symposium on Security and Privacy*, pages 122–136, Oakland, Californie, mai 1992.

[Sat89] Satyarayanan. Integration Security in a Large Distributed Systems. *ACM Transactions on Computer Systems*, 3(7):247–280, août 1989.

[SFS77] R.J. Swan, S.H. Fuller, and D.P. Siewiorek. Cm* - A Modular Multi-Microprocessor. In *Proc. of AFIPS Conf.*, pages 637–644, Montvale, 1977. vol 46, AFIPS Press.

[Str92] B. Stroustrup. *Le langage C++*. Addison-Wesley, 1992.

[Sut86] D. Sutherland. A Model of Information. In *Proc. of the 1986 IEEE Symposium on Security and Privacy*, pages 48–69, Oakland, Californie, mai 1986.

[TGC90] C-R. Tsai, V.D. Gligor, and C.S. Chandersekaran. On the Identification of Covert Storage Channels in Secure Systems. *IEEE Transactions on Software Engineering*, 16(6):569–580, juin 1990.

[TMvR86] A.S. Tanenbaum, S.J. Mullender, and R. van Renesse. Using Sparse Capabilities in a Distributed Operating Systems. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 558–563, Cambridge, Massachusetts, mai 1986.

[TNI87] Trusted Network Interpretation of the Trusted Computer Security Evaluation Criteria. National Computer Security Center, décembre 1987.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollak. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, juin 1974.

[Wei92] C. Weissman. BLACKER: Security for the DDN. Examples of A1 Security Engineering Trades. In *Proc. of the 1992 IEEE Symposium on Security and Privacy*, pages 286–292, Oakland, Californie, mai 1992.

[Woo87] J.P.L. Woodward. Exploiting the Dual Nature of Sensitivity Labels. In *Proc. of Symposium on Research in Security and Privacy, IEEE Computer Society Press*, pages 23–30, Oakland, Californie, 1987.

Thèse de doctorat de Vincent Nicomette

"La protection dans les systèmes à objets répartis"

Résumé :

La protection des systèmes répartis est un problème complexe : en quelles entités du système peut-on avoir confiance et étant donné cette confiance, comment assurer la protection du système global ? L'approche adoptée dans cette thèse consiste à combiner d'une part une gestion globale et centralisée des droits d'accès aux objets persistants du système par un serveur d'autorisation et d'autre part une protection locale par un noyau de sécurité sur chaque site du système réparti. Ce noyau contrôle les accès à tous les objets locaux (persistants ou temporaires) et a de plus la responsabilité de la gestion des droits d'accès aux objets temporaires locaux.

Un schéma d'autorisation est développé pour une telle architecture. Ce schéma est élaboré dans le cadre de systèmes composés d'objets répartis (au sens de la programmation orientée-objets). Il permet de respecter au mieux le principe du moindre privilège, définit de nouveaux droits facilement administrables (appelés droits symboliques), et un nouveau schéma de délégation de droits. Ce modèle est utilisé dans le cadre d'une politique de sécurité discrétionnaire et dans le cadre d'une politique de sécurité multiniveau. Pour cela, un modèle de sécurité multiniveau adapté au modèle objet est développé et présenté dans cette thèse. Un exemple d'implémentation de ce schéma d'autorisation est enfin détaillé.

Mots-Clés: sûreté de fonctionnement, sécurité, protection, contrôle d'accès, capacités, délégation, coupons, modèle objet.

"Protection in distributed object systems"

Abstract :

Protection in distributed systems is a complex problem: which entities of a distributed system can be trusted, and according to this trust, how can the whole system be protected ? The approach adopted in this thesis consists in distinguishing two levels of protection : a global protection by means of a centralized authorization server and a local protection on each site of the system by means of a security kernel. The authorization server has the responsibility of managing all access rights to persistant entites of the system while each security kernel controls all accesses to local objects (either transient or persistent) and is furthermore responsible for managing access rights for local transient objects.

An authorization scheme for distributed object systems is presented ("object" here refers to the object-oriented programming notion). This scheme allows the least privilege principle to be strictly respected, defines new access rights called symbolic rights and a new scheme of privilege delegation. This authorization scheme is described in the context of a discretionnary security policy and in the context of a multilevel security policy. A multilevel security model adapted to the object oriented programming paradigm is developped and presented in this thesis. An example of an implementation of this authorization scheme is finally detailed.

Keywords : dependability, security, protection, access control, capabilities, delegation, vouchers, object model.