Protection dans les architectures de systèmes flexibles

Christophe Rippert

UNIVERSITÉ JOSEPH FOURIER

THÈSE

pour obtenir le grade de

Docteur de l'Université Joseph Fourier — Grenoble 1

École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique
spécialité Informatique, Systèmes et Communication

présentée et soutenue publiquement par

Christophe RIPPERT

le 13 octobre 2003

Protection dans les architectures de systèmes flexibles

Thèse préparée au sein du laboratoire LSR-IMAG, projet SARDES

sous la direction du Pr. Sacha KRAKOWIAK

JURY

Pr. Jacques MOSSIÈRE — Président
Institut National Polytechnique de Grenoble

Dr. Bruno d'AUSBOURG — Rapporteur
Office National d'Études et de Recherches Aérospatiales

Pr. Bertil FOLLIOT — Rapporteur
Université Pierre et Marie Curie — Paris VI

Dr. Jean-Bernard STEFANI — Examinateur
Institut National de Recherche en Informatique et en Automatique

Pr. Sacha KRAKOWIAK — Directeur
Université Joseph Fourier — Grenoble I

Table des matières

Résumé

Le but de ce travail est de montrer qu'il est possible de concilier sécurité et flexibilité dans un noyau de système d'exploitation. Nous démontrons qu'il est possible de garantir à la fois la flexibilité et la sécurité d'un système, en séparant la gestion de la politique de sécurité des outils servant à sa mise en œuvre. Notre travail se découpe en deux partie. Dans la première nous proposons des outils élémentaires de protection ayant pour rôle de protéger le système contre certains types d'attaques choisis tout en restant totalement indépendants de la politique de sécurité mise en œuvre dans le système. Dans la deuxième partie nous présentons le canevas logiciel sécurisé que nous avons implanté, et notamment le composant chargé de la gestion des politiques de sécurité que nous appelons gestionnaire de sécurité. Nos résultats sont validés sur des exemples mettant en évidence la flexibilité de la protection dans le système.

Mots-clés : Systèmes d'exploitation, protection, flexibilité.

Abstract

This work aims at proving that security and flexibility can coexist in an operating system kernel. We show that both security and flexibility can be guaranteed by separating the management of the security policy from the tools used to enforce it. Our work is composed of two parts. In the first part, we propose elementary protection tools which protect the system against selected types of attacks while remaining completely independent from the security policy. In the second part, we describe the security software framework we have implemented, including the component in charge of managing security policies which we call the security manager. Our results are validated on examples showing the flexibility of protection in the system.

Keywords : Operating systems, protection, flexibility.

Remerciements

L'auteur tient à remercier toutes les personnes ayant contribué à la réussite de ce travail de thèse :

- Je remercie particulièrement le Professeur Sacha Krakowiak pour ses conseils et son encadrement tout au long de ces trois années.

- Merci également au Docteur Bruno d'Ausbourg et au Professeur Bertil Folliot d'avoir évalué ce travail et pour leurs remarques ayant permis de l'améliorer.

- Merci aussi au Professeur Jacques Mossière d'avoir accepté de présider mon jury et pour les nombreuses corrections apportées à mon rapport.

- Merci au Docteur Jean-Bernard Stefani de s'être autant impliqué dans ce travail dont il a été l'initiateur.

- Merci à tous les membres des projets SIRAC et SARDES que j'ai cotoyé au cours des 4 années que j'ai passé à l'INRIA Rhône-Alpes pour la bonne ambiance qu'ils ont contribué à faire régner.

- Je remercie particulièrement le Professeur Roland Balter qui m'a accueilli dans son laboratoire et donné le goût à la recherche.

- Merci enfin à Aurélien, Baz, Cécile, Guillaume, Jeff, Julie, Laurent, Manu, Marek, Mathieu, Matthieu, Nono, Oussama, Renaud, Simon, Takoua, Vania et Vivien pour leur camaraderie si sympathique, ainsi qu'à Aline pour sa gentillesse et son optimisme indéfectible.

Chapitre 1

Introduction

1.1 Contexte du travail

Ce travail a été réalisé au sein du projet SARDES. SARDES est un projet INRIA et une équipe de recherche du laboratoire LSR-IMAG (CNRS, INPG, UJF), localisée à l'INRIA Rhône-Alpes. SARDES est le successeur des projets/laboratoires SIRAC et Bull-IMAG.

Le projet SARDES s'inscrit dans la perspective de l'émergence d'un environnement global de traitement de l'information (informatique ubiquitaire, informatique globale) dans lequel la plupart des objets physiques qui nous entourent seront équipés de processeurs de traitement de l'information, et interconnectés par le biais de réseaux très divers (de l'internet planétaire au pico-réseau spontané).

Le projet SARDES a pour objectif principal l'étude de l'architecture et la construction d'infrastructures logicielles réparties pour cet environnement, caractérisé par une très grande taille, une très grande hétérogénéité, et par une nature très dynamique. Pour la construction de tels systèmes, sûrs et hautement adaptables, le projet se propose d'exploiter de manière systématique des techniques de réflexion et de construction par composants.

1.2 Terminologie

Il convient tout d'abord de préciser le sens des principaux termes employés dans ce document.

1.2.1 Termes liés à la sécurité

La sécurité est la propriété d'un système garantissant à ses utilisateurs une qualité de fonctionnement acceptable. Cette notion est proche dans ses objectifs de la fiabilité. On parle cependant en général d'attaques contre la sécurité d'un système et de pannes qui compromettent sa fiabilité. On peut différencier les deux en notant qu'une

attaque est une atteinte volontaire au bon fonctionnement du système effectuée par un utilisateur malveillant, alors qu'une panne résulte en général d'une erreur de programmation involontaire du concepteur du système. La sécurité est un domaine très vaste qui englobe de nombreuses thématiques, comme par exemple la protection et la cryptologie.

La protection est selon la définition de B.W. Lampson [Lam71] « l'ensemble des mécanismes permettant de contrôler l'accès d'un programme à des objets du système ». Cette definition englobe en fait trois notions distincts : la confidentialité qui consiste à empêcher un utilisateur indiscret d'accéder à des données privées, l'intégrité qui permet de garantir que des données ne seront pas modifiées ou détruites par un utilisateur n'en ayant pas le droit, et la garantie de la qualité de service qui permet d'assurer qu'aucun utilisateur ne pourra monopoliser les ressources du système dans le but d'en dégrader les performances pour les autres utilisateurs.

L'authentification est le mécanisme permettant de garantir qu'un sujet (c'est à dire un utilisateur ou un programme s'exécutant pour le compte d'un utilisateur) est bien qui il prétend être. Ce mécanisme est essentiel pour permettre la définition des droits d'accès des différents sujets et leur mise en œuvre. Il s'agit d'un problème complexe, notamment si l'on permet à un sujet de « donner » volontairement son identité à un autre sujet. Dans ce cas, il est souvent impossible pour le système de se rendre compte de la supercherie. Par exemple, un utilisateur possédant le *login* et le mot de passe d'un autre utilisateur sous Unix pourra se connecter sous cette identité sans que le système puisse faire la différence.

Une politique de sécurité est un ensemble de règles régissant les droits des différents sujets et objets du systèmes les uns par rapport aux autres. Une politique de protection regroupe donc l'ensemble des droits d'accès des sujets vers les objets du système. Il s'agit donc simplement d'une expression de la matrice de Lampson [Lam71], à laquelle on ajoute des droits de modification (c'est à dire les droits précisant quels composants peuvent modifier la politique de protection).

1.2.2 Termes liés à la flexibilité

L'adaptabilité est la propriété d'un système capable d'évoluer au cours de son exécution. Cette notion très générale inclue notamment la capacité d'ajouter dynamiquement de nouvelles fonctionnalités à un système et de modifier les paramètres des services existant pour changer leur comportement.

La flexibilité est la propriété d'un système dont l'architecture a été conçue de façon à favoriser les modifications dynamiques du système. La notion de dynamicité est importante car il est toujours possible de changer le code source d'un système et de le

recompiler[1] alors que modifier un système en cours d'exécution est souvent beaucoup plus difficile voir impossible pour les systèmes non-flexibles. Un système flexible est donc adaptable, mais un système adaptable n'est pas forcément flexible puisque le système peut être monolithique et fournir des primitives d'adaptation intégrées au noyau. La notion de flexibilité est en général liée à l'architecture du système.

La reconfiguration d'un système consiste à modifier dynamiquement ses paramètres d'exécution. Cette notion est parfois employée comme synonyme de l'adaptabilité, mais on s'accorde le plus souvent sur le fait qu'un système reconfigurable ne permet que de modifier des fonctionnalités existantes, alors qu'un système adaptable permet d'ajouter dynamiquement de nouveaux services.

1.3 Problématique

La protection a toujours été un problème majeur dans les systèmes d'exploitation multi-utilisateurs, depuis les tout premiers prototypes de Multics [CV65] jusqu'aux dernières versions de Windows [BSS01]. Qu'il s'agisse de protéger des données confidentielles contre des utilisateurs indiscrets, d'empêcher qu'un utilisateur mal-intentionné ne modifie ou détruise des données qui ne lui appartiennent pas, ou encore de garantir une bonne répartition des ressources du système pour éviter qu'une application ne les monopolise toutes, les concepteurs de systèmes ont toujours tenté de trouver des parades aux risques encourus en anticipant les actions des utilisateurs malveillants. De nombreux travaux ont été conduits à ce sujet, et bien que le domaine de la protection soit en perpétuelle évolution, car on découvre de nouvelles failles et de nouveaux types d'attaques tous les jours, certaines techniques de protection des systèmes sont aujourd'hui considérées comme classiques.

Les systèmes d'exploitation n'ont cessé d'évoluer en fonction des possibilités offertes par un matériel de plus en plus performant et des besoins d'utilisateurs toujours plus exigeants. En plus des caractéristiques essentielles telles que la fiabilité et les performances, les systèmes modernes se doivent notamment d'être capables de s'adapter à leur environnement et aux exigences des utilisateurs pour fournir à chaque instant un service de qualité optimale. C'est pour satisfaire cette exigence d'adaptabilité qu'ont été développés les systèmes flexibles, permettant à l'administrateur de faire évoluer dynamiquement le noyau même du système pour y ajouter des fonctionnalités ou modifier celles déjà en place, par exemple par un système de modules comme dans les noyaux Linux [Aiv02] récents. Parallèlement ont été développées des architectures de systèmes permettant de construire des systèmes à la carte, exploitant au mieux les possibilités du matériel sous-jacent et n'intégrant que les services véritablement utiles, à la différence des premiers systèmes qui intégraient systématiquement toutes les abstractions disponibles. On peut citer le projet

[1] En supposant bien sûr que le code source du système est disponible...

OSKit [FBB+97] qui fournit une bibliothèque de services librement assemblables et l'architecture exo-noyau [EKO95] qui définit un modèle de construction de systèmes n'intégrant aucun service dans le noyau afin de laisser au programmeur système une liberté totale sur les abstractions qu'il désire inclure dans son système.

L'architecture de systèmes THINK [Fas01] [FSLM02] s'inscrit dans la continuité de cette volonté de flexibilité. Tout comme OSKit, elle fournit une bibliothèque de composants systèmes pouvant être assemblés à la guise du programmeur afin de construire des systèmes à la carte. Elle intègre de plus l'orientation minimaliste de l'exo-noyau en étant basée sur un noyau minimal n'intégrant aucune abstraction système. Enfin, son canevas logiciel basé sur un modèle à composants supporte l'adaptation dynamique du système et permet le chargement de composants en cours d'exécution. THINK est donc un outil particulièrement intéressant pour qui désire construire un système flexible et optimal.

Malheureusement, les notions de protection et de flexibilité sont souvent considérées comme antagonistes dans de nombreux systèmes. En effet, beaucoup de concepteurs de systèmes estiment que permettre à un système d'évoluer, notamment en y ajoutant de nouvelles fonctionnalités, engendre forcément de nouvelles vulnérabilités qui n'avaient pas été prises en compte lors de la conception du système et donc contre lesquelles aucune protection ne peut être mise en place a priori. Par exemple, l'introduction des modules dans les noyaux Linux récents pose de gros problèmes de sécurité puisqu'il n'existe à l'heure actuelle aucune vérification de l'innocuité du module lors de son chargement, ni de son origine. La seule protection réside dans le fait qu'il faut être connecté en tant qu'administrateur du système pour pouvoir charger le module, ce qui n'est pas une protection efficace en soi car il est facile de tromper l'administrateur en insérant un cheval de Troie dans un pilote de périphérique par exemple[2].

Réciproquement, augmenter la sécurité d'un système revient le plus souvent à restreindre la liberté de l'administrateur ou des utilisateurs en interdisant l'utilisation des services dont des utilisateurs malveillants pourraient potentiellement se servir pour attaquer le système. Dans l'exemple précédent, on pourrait aisément enlever la gestion des modules dans le noyau de Linux pour supprimer le risque d'introduction de code malveillant. Cela diminuerait grandement la flexibilité du noyau en obligeant l'administrateur à le recompiler et à redémarrer la machine à chaque fois qu'il souhaite le faire évoluer. Plus généralement, même s'il essaie d'éviter d'interdire purement et simplement tous les mécanismes pouvant potentiellement être utilisés de façon malveillante, le concepteur d'un système a souvent tendance a imposer ses vues en matière de sécurité notamment dans la programmation des services fondamentaux du système. Par exemple, l'isolation des pro-

[2]Les défenseurs du système de modules de Linux rétorquent à cela que c'est la responsabilité de l'administrateur de vérifier l'origine et le fonctionnement des modules qu'il introduit dans le noyau, mais en pratique il est irréaliste de penser qu'un administrateur système lira en détail le code source de chaque module qu'il ajoute à son système.

cessus dans Linux sur Pentium est complètement dépendante du mécanisme de contrôle d'accès à la mémoire paginée et ne peut être changé sans reprogrammer entièrement le gestionnaire mémoire et une partie importante du système qui en dépend directement.

1.4 Objectif

Le but de ce travail est de montrer qu'il est possible de concilier flexibilité et sécurité dans un noyau de système d'exploitation. On doit tout d'abord être capable de garantir que les mécanismes de protection intégrés au système ne réduise pas sa flexibilité. Dans le cas d'un système existant auquel on ajoute des mécanismes de protection, on doit s'assurer qu'on ne compromet pas la flexibilité originale du système ce faisant. Par exemple, interdire l'utilisation des modules dans un noyau Linux compromet la flexibilité du système puisqu'on empêche l'ajout ou le retrait dynamique de pilotes ou de services systèmes. Dans le cas d'un système en construction, on doit garantir que les mécanismes de protection que l'on intègre n'imposeront pas de contrainte sur le développement des autres parties du noyau. Typiquement, imposer un mécanisme d'isolation des processus basé sur la segmentation fournie par le processeur impose de mettre en place des mécanismes de communication inter-processus. Un tel choix de gestion de la mémoire a des conséquences importantes sur le développement des autres services systèmes qui dépendent directement de la façon dont est gérée la mémoire. On doit donc dans tous les cas s'assurer que les choix que l'on fait en matière de protection auront le moins d'impact possible sur la liberté du programmeur système à développer le reste de son système comme il l'entend.

Il est également nécessaire de garantir que les mécanismes de protection implantés dans le système sont eux même flexibles. On entend par là que ces mécanismes ne doivent pas imposer au programmeur système de choix quant à la politique de protection qu'il souhaite mettre en œuvre dans le système. Cela impose notamment d'assurer une séparation totale entre la gestion de la politique de protection et les outils servant à la mettre en œuvre, c'est à dire les mécanismes de protection implantés dans le système. Cette idée n'est pas nouvelle puisqu'elle a été proposée pour la première fois dans le système Hydra [WLP75] [CJ75] [LCC+75] en 1975 mais elle n'a malheureusement pas été exploitée par les concepteurs de systèmes jusqu'à très récemment dans le prototype expérimental DTOS [OFSS96].

Pour s'assurer que cette séparation de la gestion de la politique des outils servant à sa mise en œuvre est effectivement possible en pratique, on doit d'abord montrer qu'il est possible de fournir des outils de protection réellement indépendants de la politique de sécurité. Cela passe par la spécification et l'implantation de mécanismes offrant une protection contre des attaques choisies et l'évaluation de leur flexibilité. Cette évaluation doit être réalisée de façon concrète en utilisant les outils proposés dans des systèmes réalistes dans lesquels on fera évoluer la politique de protection pour montrer que l'outil

n'a pas besoin d'être modifié pour prendre en compte cette évolution.

Parallèlement, on doit montrer qu'il est possible de spécifier des politiques de protection flexibles, c'est à dire pouvant être modifiées dynamiquement. Cette flexibilité doit être évaluée de façon similaire à celle des outils, en testant des modifications dynamiques de la politique dans des systèmes fonctionnels. De plus, on doit s'assurer que la mise en œuvre de la politique choisie via les outils de protection proposés se fera elle-même de façon sécurisée, c'est à dire concrètement qu'il n'est pas possible pour un composant d'ignorer la politique définie dans le système en « court-circuitant » les mécanismes utilisés pour la mettre en œuvre. L'architecture du système doit garantir que les interactions entre les différents modules qui composent le système sont réalisées de façon sûre tout en gardant leur flexibilité initiale.

Enfin, on doit être capable de valider ces résultats en les mettant en œuvre dans des situations réalistes. La validation réalisée doit permettre d'évaluer l'indépendance des outils vis-à-vis de la politique de sécurité ainsi que la flexibilité de l'architecture logicielle. Il est également nécessaire de s'assurer que les mécanismes sont suffisament performants pour être utilisables dans des systèmes réalistes. La performance n'est pas la priorité de ce travail, puisqu'on privilégie la flexibilite et la securité des mécanismes, mais il est évidemment nécessaire que ces mécanismes ne pénalisent pas l'exécution du système au point de le rendre inutilisable.

1.5 Mise en œuvre

Afin de démontrer qu'il est possible de concilier sécurité et flexibilité dans un système d'exploitation, nous avons choisi de partir de systèmes flexibles existants et de les sécuriser en montrant qu'on ne réduit pas la flexibilité initiale ce faisant. Pour cela, nous utilisons l'architecture de systèmes flexibles THINK pour construire les systèmes dans lesquels nous implantons des mécanismes de protection. L'intérêt de s'appuyer sur l'architecture THINK est qu'elle facilite la construction de systèmes flexibles grâce à sa bibliothèque de services, ce qui permet de construire des noyaux fonctionnels en quelques minutes si l'on utilise les composants fournis. De plus, cette architecture n'est actuellement pas du tout sécurisée ce qui permet donc d'expérimenter nos propres mécanismes sans contrainte imposée par l'existant.

Dans le but de montrer qu'il est possible de séparer la gestion de la politique de sécurité des outils permettant sa mise en œuvre, nous proposons tout d'abord un ensemble d'outils élémentaires de sécurité. Le terme élémentaire signifie que chaque outil est conçu de la façon la plus indépendante possible des autres composants et assure la protection du système contre un type d'attaque précis. Cette architecture modulaire est un facteur de flexibilité intéressant si l'on souhaite remplacer des outils ou simplement les décharger du

système. Pour chaque outil, il est important de préciser le type d'attaque contre lequel il protège le système avant de détailler son implantation dans un système fonctionnel. A noter que les outils proposés ne sont pas spécifiques à l'architecture THINK puisque notamment ceux concernant la protection du réseau ont été implantés dans le gestionnaire TCP/IP de Linux d'où est extrait une partie du code du gestionnaire réseau intégré dans la bibliothèque de services systèmes de THINK. Il est également intéressant d'évaluer les performances des outils proposés, bien que l'efficacité ne soit pas notre priorité.

La deuxième partie de ce travail consiste à montrer qu'il est possible de concevoir une architecture de système où la gestion de la politique de sécurité est dissociée des outils assurant sa mise en œuvre. Pour cela, on part de la spécification du canevas logiciel de l'architecture THINK pour en proposer une nouvelle implantation sécurisée. Ce canevas sécurisé doit intégrer un composant gérant les politiques de sécurité avec lequel les outils élémentaires dialoguent lors des prises de décision. En plus de permettre de piloter les outils élémentaires via le gestionnaire de sécurité, on montre que le canevas permet des interactions sécurisées entre tous les composants du système.

Concernant la spécification des politiques de sécurité, un travail doit être réalisé pour faciliter la tâche du programmeur système. Entre une approche purement statique (c'est à dire une politique programmée dans le gestionnaire de sécurité et ne pouvant être modifiée) et une gestion complètement dynamique comme celle fournie par le langage Prolog qui n'est pas réaliste dans un noyau de système, on montre qu'on peut trouver un compromis permettant de simplifier la spécification des politiques sans compromettre les performances du système. A noter que cette étude n'a pas la prétention d'être exhaustive car le problème de la spécification et de la gestion dynamique de politiques (de sécurité ou autres) dans un système est beaucoup trop vaste pour être traité intégralement ici.

Enfin, ces résultats sont validés sur le plan de la flexibilité. On montre que les outils sont réellement indépendants de la politique de sécurité en faisant évoluer celle-ci dynamiquement dans un système et en montrant que l'outil lui-même n'a pas besoin d'être modifié. On montre que le canevas sécurisé peut être utilisé pour mettre en œuvre d'autres types de protection que le simple contrôle d'accès. Enfin, on valide l'utilisation des outils et du canevas via un exemple de système concrètement utilisable.

1.6 Plan

Le chapitre 2 présente la problématique de la protection dans les systèmes d'exploitation. On y décrit d'abord les principales techniques couramment utilisées pour garantir le contrôle d'accès dans les systèmes classiques. Puis on détaille les principaux algorithmes de gestion de ressources en se focalisant sur quelques ressources centrales d'un ordinateur moderne, à savoir le processeur, le disque dur et le réseau.

Le chapitre 3 décrit les principales architectures de systèmes d'exploitation. On présente les architectures classiques développées au cours de l'évolution des systèmes, puis les prototypes de recherches les plus récents. On met en évidence les intérêts et les inconvénients de ces architectures du point de vue de la protection et de la flexibilité. Enfin, on présente brièvement l'architecture de systèmes flexibles THINK.

Le chapitre 4 présente les outils élémentaires de sécurité que nous avons spécifiés et implantés. On se base sur la proposition de mécanismes de défense de trois ressources essentielles du système pour montrer qu'il est possible de spécifier et d'implanter dans un système des outils de protection qui restent indépendants de la politique de sécurité choisie et n'imposent aucune contrainte au programmeur système concernant la mise en place de cette politique. Nos travaux sur l'isolation mémoire, l'ordonnancement des requêtes d'accès au disque dur, et la protection du gestionnaire de réseau sont des exemples d'outils pouvant être utilisés pour défendre le système contre des attaques ciblées sans mettre en cause la flexibilité du dit système.

Le chapitre 5 détaille la sécurisation du canevas logiciel de l'architecture THINK. Le but de ce travail est de montrer qu'il est possible de sécuriser une architecture flexible existante sans compromettre sa flexibilité. On analyse tout d'abord les déficiences de l'architecture THINK en matière de sécurité dans son état actuel et on décrit ensuite les modifications que nous avons effectuées afin de la sécuriser. On présente aussi dans ce chapitre quelques idées concernant l'expression de la politique de sécurité dans un système. Il ne s'agit pas ici de fournir une solution complète à ce problème qui est trop vaste pour être traité dans le cadre de ce travail, mais de proposer des solutions spécifiques à l'architecture THINK et à notre implantation de son canevas sécurisé.

Le chapitre 6 présente plusieurs applications des résultats présentés aux chapitres 4 et 5. Ces applications permettent d'évaluer la flexibilité des techniques proposées et montrent qu'on est capable de préserver la flexibilité du système tout en ajoutant de la sécurité. On présente également un canevas de gestion de ressources simple dans l'architecture THINK qui illustre la mise en œuvre d'un canevas logiciel flexible dans un but qui dépasse le contrôle d'accès pour prendre en compte la gestion de la qualité de service dans le système. Enfin, on valide ces travaux sur un exemple réaliste d'utilisation dans le cadre d'un système d'exploitation flexible et sécurisé pour les nœuds d'un réseau actif.

Chapitre 2

Protection dans les systèmes d'exploitation

2.1 Introduction

Si on se réfère à la définition de B.W Lampson [Lam71], la protection dans un système consiste à se protéger contre trois types d'attaques : l'altération ou la destruction de données, la lecture non autorisée de données confidentielles, la consommation excessive des ressources du système dans le but de dégrader le service offert. La réponse aux deux premiers types d'attaque est connue sous le nom de contrôle d'accès (de l'anglais *access control*). Pour le troisième type, on parle de garantie de la qualité de service (de l'anglais *Quality of Service*). Nous présentons ci-dessous ces notions ainsi que les principales techniques utilisées pour les mettre en œuvre. On notera que l'on ne parle ici que de protection au sens strict du terme, tel que défini par Lampson. La sécurité de façon plus générale englobe bien d'autres notions tout aussi importantes, comme l'authentification par exemple. On emploiera donc le terme de protection pour désigner les mécanismes garantissant le contrôle d'accès et la gestion de la qualité de service, et le terme de sécurité pour désigner la propriété globale caractérisant un système sûr.

2.2 Contrôle d'accès

Le contrôle d'accès vise à réglementer les accès aux différentes ressources du système en fonction de l'utilisateur cherchant à effectuer l'accès (utilisateur s'entend ici au sens utilisateur de la ressource, et peut aussi bien désigner une personne physique utilisant le système qu'un processus). On liste traditionnellement les droits d'accès aux ressources sous forme d'une matrice d'accès, comprenant en colonnes les ressources protégées, et en lignes les utilisateurs. La figure 2.1 donne un exemple d'une telle matrice pour la gestion

des droits d'accès à des fichiers sous Unix.

	`donnees.dat`	`programme.exe`	`repertoire_prive`
Alice	RW	RWX	RWX
Groupe d'Alice	R	RX	∅
Tout le monde	R	RX	∅

FIG. 2.1 – Matrice d'accès pour les fichiers d'Alice

La lettre 'R' désigne traditionnellement le droit de lecture, 'W' le droit d'écriture, et 'X' le droit d'exécution.

Une telle matrice peut être lue de deux façons. Si on la lit en colonnes, on retrouve la notion de liste de contrôle d'accès (de l'anglais *Access Control List*), utilisée dans de nombreux systèmes d'exploitation. Dans l'exemple de la figure 2.1, on obtient les droits d'accès Unix à chaque fichier en lisant la matrice en colonnes, par exemple `donnees.dat` : `rw-r--r`. A contrario, si on lit la matrice ligne par ligne, on retrouve la notion de liste de capacités (de l'anglais *capability list*), qui détaille tous les droits d'accès que possède un utilisateur donné. L'avantage des listes de contrôle d'accès est qu'elles permettent d'avoir une vue immédiate de toutes les autorisations associées à une ressource alors que les capacités permettent au contraire d'accéder rapidement à tous les droits que possède un utilisateur donné. Le choix de la technique se fera donc selon la vue que l'on désire avoir, et les deux techniques peuvent également être combinées.

L'implantation de la notion de capacité a donné lieu à de nombreuses propositions depuis la spécification originale par Dennis et Van Horn [DVH66], notamment en ce qui concerne le contrôle d'accès à la capacité elle-même, pour éviter qu'un processus malveillant ne modifie une capacité pour s'ajouter des droits. On distingue trois grandes classes d'implantation des capacités. Les capacités dites « marquées » sont des capacités stockées dans des mots mémoire reservés aux capacités. Cette implantation suppose donc un support du matériel sous-jacent, qu'on appelle alors une machine à capacités. Les capacités dites « confinées » sont quant à elles stockées dans une zone mémoire non accessibles aux utilisateurs, typiquement dans la zone système. Enfin, les capacités « chiffrées » utilisent des techniques de cryptographie et de somme de contrôle pour garantir l'intégrité et la confidentialité de la capacité. Des travaux concernant les capacités ont notamment été réalisés par Hagimont [Hag98] et Jensen [Jen99].

2.2.1 Politiques de contrôle d'accès

On présente dans cette section les politiques de contrôle d'accès traditionnellement mises en œuvre dans les systèmes d'exploitation.

2.2.1.1 Politique discrétionnaire

Une politique discrétionnaire se base sur l'association à chaque utilisateur de droits d'accès pour chaque ressource du système. Ce type de politique est la plus couramment employée et se retrouve dans de nombreux systèmes d'exploitation, par exemple dans la gestion des droits d'accès aux fichiers d'un système Unix comme dans l'exemple ci-dessus. On peut dissocier deux types de politiques discrétionnaires, les politiques fermées qui consistent à interdire l'accès à une ressource à tout le monde par défaut, et les politiques ouvertes qui supposent que les ressources sont par défaut accessibles à tout le monde. Dans le premier cas, on doit spécifier explicitement les utilisateurs qui ont accès à la ressource, alors que dans le deuxième, on liste les utilisateurs qui ne doivent pas avoir accès à la ressource. Ces deux types de politiques peuvent être combinées, bien que cela complique la gestion en général. Le contrôle d'accès aux services des machines Unix est un exemple de politique discrétionnaire ouverte. Lorsqu'un client émet une requête à un service d'une machine Unix, le système vérifie si l'association client/service est listée dans le fichier `/etc/hosts.allow`. Si c'est le cas, l'accès est permis. Sinon, le système vérifie si le couple client/service est listé dans le fichier `/etc/hosts.deny`. Si c'est le cas, l'accès est refusé. Sinon, l'accès est autorisé.

Les politiques discrétionnaires ont cependant un problème majeur qui est l'absence de contrôle possible sur la dissémination des ressources. En effet, rien n'empêche l'utilisatrice Alice qui a le droit d'accéder à la ressource R, de copier cette ressource et de la rendre disponible à l'utilisateur Bob qui lui n'a pas le droit d'accéder à la ressource R. L'exemple d'un fichier contenant des données confidentielles qu'Alice recopie pour les donner à Bob illustre cette difficulté.

2.2.1.2 Politique mandataire

Une politique mandataire est mise en œuvre par une autorité qui associe à chaque utilisateur un niveau de confiance et à chaque ressource un niveau de confidentialité. Ces niveaux sont utilisés pour déterminer les actions que peut effectuer un utilisateur sur une ressource donnée. On définit une relation d'ordre entre les différents niveaux, par exemple dans un contexte militaire on pourrait utiliser les niveaux Top Secret, Secret, Confidentiel et Public, et définir la relation d'ordre : TS > S > C > P. Les accès sont réglementés par les deux principes suivants :

Lecture descendante : le niveau de confiance d'un utilisateur doit être supérieur ou égal au niveau de confidentialité de la donnée qu'il veut lire.

Ecriture ascendante : le niveau de confiance d'un utilisateur doit être inférieur ou égal au niveau de confidentialité de la donnée qu'il veut écrire.

Le principe de lecture descendante est immédiatement compréhensible, puisqu'il spécifie simplement qu'un utilisateur ne peut pas lire une donnée plus confidentielle que ce qu'il est autorisé à lire. Le principe d'écriture ascendante permet quand à lui d'interdire à un utilisateur de décroître le niveau de confidentialité d'une donnée. Par exemple, l'utilisatrice Alice de niveau de confiance TS pourrait vouloir lire une donnée TS et l'écrire dans un fichier de niveau P afin de la rendre accessible à l'utilisateur Bob qui est lui de niveau P. Un exemple bien connu de modèle de politique mandataire est le modèle de Bell et LaPadula [BL73].

On notera qu'en pratique une politique discrétionnaire est souvent associée à une politique mandataire. Le principe d'écriture ascendante seul permet en effet à un utilisateur de modifier ou détruire des fichiers qui ne lui appartiennent pas tant qu'ils sont de niveaux supérieurs au sien. C'est alors le rôle d'une politique discrétionnaire d'empêcher ce type d'abus.

2.2.1.3 Politique par rôles

Une politique par rôles définit un ensemble de profils d'utilisateur (les « rôles » en question) auxquels sont attachés les autorisations nécessaires à l'accomplissement des tâches associées à chaque profil. Cette notion de rôle se retrouve par exemple dans les systèmes Unix où on différencie le profil de l'administrateur système (`root`) du profil des utilisateurs normaux. L'intérêt de cette notion de rôle est qu'un même utilisateur peut endosser tel ou tel rôle selon les tâches qu'il a à accomplir, comme dans un système Unix un utilisateur peut s'identifier en `root` lorsqu'il doit accomplir des tâches d'administration. Certaines politiques par rôles peuvent permettre à un utilisateur d'exercer plusieurs rôles simultanément si les tâches associées à chaque rôle sont disjointes, mais en général un utilisateur doit explicitement renoncer à un rôle avant d'en endosser un nouveau (par exemple sous Unix l'utilisatrice Alice exécutant la commande `su` pour acquérir le rôle de `root` perd son rôle d'Alice et devient uniquement `root`, jusqu'à ce qu'elle exécute la commande `exit` pour redevenir Alice).

L'avantage d'une politique par rôle est qu'elle permet de structurer la gestion des droits d'accès de façon à refléter une organisation humaine. Par exemple, dans une entreprise, on peut définir des rôles en fonction des différentes activités des employés, et gérer de façon flexible l'attribution et la révocation d'autorisation. Ainsi, si Alice change de service, il suffit à l'administrateur de lui donner accès au rôle correspondant à ces nouvelles attributions et de révoquer son accès à son ancien rôle pour qu'elle acquière automatiquement les autorisations dont elle aura besoin. Une telle politique permet de plus d'éviter de donner en permanence des autorisations à des utilisateurs qui n'en ont besoin que par intermittence. Dans un système Unix, l'administrateur système ne se connecte en `root` que lorsqu'il a besoin d'effectuer des tâches d'administration, et en temps qu'utilisateur

normal le reste du temps. Cela permet notamment d'éviter qu'une erreur de manipulation ait des conséquences désastreuses pour le système.

2.2.2 Isolation mémoire

L'isolation mémoire est un mécanisme de contrôle d'accès de bas niveau présent dans la majorité des systèmes d'exploitation actuels et permettant d'implanter la notion de domaine de protection. Il est en effet nécessaire de pouvoir restreindre l'accès à certaines zones mémoire contenant du code ou des données critiques pour le fonctionnement du système. De plus, dans les systèmes multi-processus, on ne peut tolérer qu'un processus applicatif puisse accéder au code ou aux données d'un autre processus. Un processus malveillant pourrait par exemple récupérer des données confidentielles, et un processus défectueux pourrait écraser le code d'un autre processus à cause d'une erreur de pointeur par exemple. Aussi, des mécanismes variés ont été mis au point pour permettre de définir des zones mémoire isolées les unes des autres. Nous présentons ci-dessous les principaux mécanismes utilisés actuellement.

2.2.2.1 Isolation matérielle

L'isolation matérielle consiste à utiliser les mécanismes de vérification des droits d'accès fournis par le processeur pour isoler les processus dans des espaces d'adressage différents. On rappelle ci-dessous les principales techniques de gestion de la mémoire et leur mise en œuvre.

L'unité de gestion de la mémoire Ce circuit, intégré au processeur, a la charge d'effectuer tous les accès mémoire. Il peut en outre assurer d'autres fonctions comme nous le voyons ci-après.

Notion de mémoire paginée La pagination mémoire permet de définir des zones mémoire de taille fixe, appelées pages, et pouvant avoir des caractéristiques particulières. Par exemple, si l'unité de gestion de la mémoire permet de définir différents types de droits d'accès (lecture seule, lecture et écriture, exécution seule, ...) on peut associer des droits a une page donnée et l'unité de gestion de la mémoire vérifiera que ces droits sont bien respectés lors de chaque accès mémoire à cette page. Ces informations sont stockées dans la table des pages, qui est un tableau en mémoire vive qui répertorie les pages du systèmes en leur associant l'adresse physique du premier octet de chacune, ainsi que leur éventuelles propriétés.

Mécanisme de va-et-vient Un mécanisme de va-et-vient (de l'anglais *swapping*) permet de libérer de la mémoire vive en recopiant le contenu des pages les moins récemment

accédées sur le disque dur. Les pages « déchargées » sont alors marquées comme telles dans la table des pages et l'espace mémoire correspondant est réaffecté à une autre page. Le mécanisme inverse a lieu lorsqu'un accès à la page déchargée est exécuté, le système recharge alors le contenu de l'ancienne page dans une zone mémoire libre. Au cas où la mémoire serait remplie, il faudrait alors décharger une autre page pour pouvoir recharger l'ancienne, ce qui impose de choisir avec soin l'algorithme de sélection des pages à décharger pour éviter de décharger une page dont on aura besoin peu de temps après.

Notion de mémoire virtuelle La mémoire virtuelle est une abstraction de la mémoire physique fournie aux processus par le processeur. Les processus utilisent donc des adresses dites virtuelles pour accéder à la mémoire, et ces adresses sont traduites de façon transparentes en adresse physique par l'unité de gestion de la mémoire lorsqu'il effectue les accès mémoire. On notera que cette notion est complètement distincte de la notion de pagination mémoire. Cependant, ces deux mécanismes sont le plus souvent utilisés en coopération, afin de combiner leurs avantages.

Le tampon de traduction transparente Le tampon de traduction transparente (de l'anglais *Translation Look-aside Buffer*[1]) est une table de correspondance entre les adresses virtuelles et les adresses physiques. Il est situé dans l'unité de gestion de la mémoire. Dans le cas où la mémoire est paginée, ce tampon est en général utilisé comme antémémoire (de l'anglais *cache memory*) de la table des pages, afin d'éviter un accès mémoire lors de la traduction, ce qui bien évidemment ralentirait le décodage et l'exécution des instructions. Cet accès est cependant nécessaire au cas où la correspondance recherchée n'est pas dans le tampon de traduction. Au cas où elle n'est pas non plus dans la table des pages, cela signifie que la page contenant l'adresse accédée n'est pas en mémoire vive et une exception est levée par le processeur, afin de permettre au système d'exploitation soit de la charger depuis le disque dur si un mécanisme de va-et-vient est utilisé, soit de neutraliser le processus émetteur sinon car il aura essayé d'exécuter un accès mémoire incorrect.

Notion de mémoire segmentée La segmentation est un mécanisme permettant la définition de plusieurs espaces d'adressage linéaires. L'intérêt de ce mécanisme est de permettre de découper les programmes et les données selon des critères logiques. Par exemple, une application peut être répartie dans deux segments, l'un contenant son code et l'autre ses données. Le segment contenant le code peut alors être protégé de façon à ne pouvoir être accédé qu'en exécution et celui contenant les données qu'en lecture et en

[1] Cette traduction sémantique a pour but de mettre en évidence le fait que la recherche de la correspondance adresse virtuelle / adresse physique dans la table est effectuée en parallèle du décodage de l'instruction dans le processeur, d'où le *look-aside* qu'on pourrait traduire par « en jetant un coup d'œil à la table pendant le décodage ».

écriture. Ce type de protection existe également avec un mécanisme de mémoire paginée mais le programmeur ne sait pas en général si une page contient du code ou des données alors qu'il connait le contenu de ses segments. La segmentation est donc un mécanisme qui doit être pris en compte par le programmeur lorsqu'il développe son application alors que la pagination est transparente de son point de vue. La segmentation facilite enfin le partage de code entre différentes applications, comme les bibliothèques partagées des systèmes Linux par exemple.

L'adressage se fait alors via des adresses virtuelles, décomposées en deux parties. La première partie, appelée adresse de segment ou adresse de base, désigne le premier octet du segment. La deuxième partie, appelé décalage, est l'adresse relative de l'octet pointé par rapport à la base du segment. Par exemple, dans le processeur 8086 d'Intel, une adresse virtuelle a la forme : `<seg:off>` où `seg` et `off` sont des entiers sur 16 bits. L'adresse physique correspondante est calculée simplement en utilisant la formule : $phys = seg \times 16 + off$. On notera cependant qu'en général dans les processeurs modernes supportant la segmentation, la correspondance entre l'adresse virtuelle et l'adresse physique n'est pas aussi simple, et passe généralement par le biais d'un tableau en mémoire vive, appelée table des descripteurs de segment pour le Pentium d'Intel par exemple, qui catalogue l'ensemble des segments alloués dans le système ainsi que les propriétés qui leur sont associées. Ainsi, pour le Pentium, l'adresse de segment est en fait un index dans la table des descripteurs de segments [Int97].

On notera que la notion de mémoire segmentée dépend de celle de mémoire virtuelle mais pas de la pagination, bien qu'en pratique ces trois mécanismes sont souvent associées. De la même façon, on remarquera que le fait de bénéficier d'un mécanisme de mémoire segmentée n'implique pas forcément que l'unité de gestion de la mémoire soit capable de gérer des droits d'accès aux différents segments. C'est le cas par exemple pour le 8086 où l'unité de gestion de la mémoire est simplifiée au possible et est juste capable d'effectuer le calcul de l'adresse physique selon la formule donnée précédemment.

Évaluation En supposant donc que le processeur sous-jacent permette la segmentation de la mémoire et l'association de droit d'accès à chaque segment, on peut définir des zones mémoire isolées les unes des autres et garantir la protection des processus les uns vis à vis des autres. Cependant, il existe des architectures qui ne fournissent pas de mécanisme de protection matérielle. C'est le cas du 8086 comme noté précédemment, mais aussi d'autres processeurs plus récents couramment utilisés dans les systèmes embarqués. En effet, ce type de systèmes, fonctionnant souvent sur batterie, se doit d'avoir une consommation électrique la plus réduite possible, et nécessite donc des processeurs simplifiés. Pour ce type de systèmes, l'isolation matérielle est donc impossible.

De plus, l'isolation matérielle entraîne un problème de performance important pour les

appels inter-processus (de l'anglais *Inter-Process Communication*). En effet, lorsqu'on veut permettre à un processus d'accéder légitimement à une zone mémoire appartenant à un autre processus, on est contraint de mettre en œuvre des mécanismes destinés à contourner l'isolation matérielle. Ces mécanismes, comme les *sockets*[2] par exemple, se révèlent en général très coûteux à l'exécution. C'est d'ailleurs un des problèmes principaux des micro-noyaux, qui se basent sur l'isolation matérielle pour protéger les services systèmes les uns des autres, ce qui est très pénalisant lorsque ces services ont besoin de communiquer fréquemment.

Enfin, l'isolation matérielle manque de flexibilité puisque les vérifications pouvant être effectuées lors des accès aux pages ou aux segments se limitent aux mécanismes fournis par les concepteurs du processeur. Il n'est donc pas possible d'ajouter de nouveaux types de vérification ou de remplacer ceux proposés par la machine. Par exemple, le programmeur système pourrait vouloir n'autoriser qu'un nombre fixé d'accès à une zone mémoire et révoquer ce droit d'accès dès que le processus source a dépassé son quota d'accès. Cette politique très simple est difficile à mettre en œuvre avec un mécanisme d'isolation matérielle car l'unité de gestion de la mémoire ne fournit en general aucun support pour compter le nombre d'accès effectué à une page ou une segment. A contrario, ce test est très simple à mettre en œuvre si la vérification est codée par le programmeur système, comme c'est le cas avec un mécanisme d'isolation logicielle comme ceux que nous présentons ci-dessous.

2.2.2.2 Isolation logicielle

A la différence d'un mécanisme d'isolation matérielle, l'isolation logicielle ne nécessite aucun support du matériel et peut fonctionner avec un processeur ne comprennant pas d'unité de gestion de la mémoire. Ce type d'isolation peut prendre plusieurs formes, dont nous détaillons les deux principales ci-dessous.

Isolation basée sur le langage Cette technique consiste à utiliser un langage offrant des propriétés de sûreté pour développer les modules. De telle propriétés incluent par exemple des vérifications sur les changements de types ou un contrôle d'accès à la compilation. En vérifiant que tout appel de fonction ou accès à une variable est autorisé lors de la compilation du module, on peut assurer une isolation flexible des composants du système. Il est cependant nécessaire d'interdire la manipulation explicite de pointeurs dans les programmes, afin d'éviter la fabrication de pointeurs permettant de court-circuiter les vérifications à la compilation. On donne dans la figure 2.2 un exemple de code écrit en C et en binaire PowerPC permettant d'effectuer illégalement un appel à une fonction interdite en fabriquant un pointeur.

[2]Ce mot anglais étant universellement utilisé dans le monde du système, nous ne chercherons pas à le traduire.

```
void evilCode {
  int instrArray[] = {
    0x480ABC03, // call forbiddenFct
    0x4E800020  // return
  };
  void (*fct)() = (void (*)()) instrArray;
  fct();
}
```

FIG. 2.2 – Un exemple simple de fabrication de pointeur

Des vérifications dynamiques peuvent également être mises en place en générant du code de vérification à la compilation du programme. La génération de code vérifiant que tous les accès à un tableau se font bien entre ses bornes est un exemple typique de ce type de vérification.

Java [GJSB00] et Modula-3 [Har92] sont des exemples bien connus de langages offrant des propriétés de sûreté intéressantes pour l'isolation des modules. Par exemple, Java rend impossible la fabrication de pointeur en cachant leur manipulation derrière la notion de référence. Il propose aussi des mécanismes de contrôle d'accès propres aux langages à objets, manifestés par les mots-clés `private`, `protected` et `public`. On notera qu'il s'agit également d'un langage destiné à être interprété (bien qu'il existe actuellement des compilateurs de programmes Java en binaire), ce qui permet un certain nombre de vérifications supplémentaires, comme nous le verrons ci-après.

Le principal inconvénient de cette technique est qu'elle impose d'avoir confiance dans la chaîne de compilation et par conséquent dans les binaires générés. En effet, il est facile à un programmeur mal intentionné de modifier un compilateur pour par exemple désactiver la vérification des appels de fonctions externes (certains compilateurs facilitent même cette tâche en proposant ces vérifications sous formes d'options de compilation désactivables). De plus, même en supposant que la chaîne de compilation est sûre, il est également assez simple de modifier un binaire après sa compilation. Des techniques de signature et de calcul de sommes de contrôle des binaires générés peuvent alors être mises en place pour garantir leur authenticité. Ce système de signature est par exemple utilisé dans Windows XP pour garantir la sécurité des pilotes de périphériques téléchargés. Mais il est souvent possible de générer de fausses signatures permettant de tromper le mécanisme de contrôle.

Isolation par injection de code Cette technique consiste à générer du code de vérification avant les accès mémoire à vérifier. Elle se rapproche des vérifications dynamiques générées par certains compilateurs, comme les vérifications des accès à un tableau par exemple, mais le terme d'injection de code implique en général un mécanisme disjoint du processus de compilation, et le plus souvent travaillant sur des binaires plutôt que sur

du code source. Le mécanisme le plus connu d'isolation par injection de code est l'isolation de faute logicielle décrite dans [WLAG93] dont nous détaillons les deux déclinaisons principales ci-dessous.

Correspondance de segments La correspondance de segments, de l'anglais *segment matching*, consiste à imiter la segmentation matérielle de la mémoire présentée plus haut en découpant de façon logicielle l'espace mémoire en segments. Pour cela, on considère les n bits de poids fort de chaque adresse mémoire comme un identificateur de segment, n étant un nombre de bits choisi en fonction du nombre de segments maximum voulu. Le code que l'on injecte avant chaque accès mémoire a donc pour rôle de vérifier que l'identificateur de segment contenu dans l'adresse de destination de l'accès correspond bien à un segment autorisé, par exemple le segment dans lequel s'exécute le code si l'on souhaite l'isoler complètement.

Bac à sable La technique du bac à sable (de l'anglais *sandboxing*) est une version optimisée de la correspondance de segments. Pour éviter une comparaison systématique pour chaque accès mémoire, le code injecté écrase l'identificateur de segment de l'adresse par l'identificateur du segment contenant le module. Ainsi, si l'identificateur initial était différent, on force l'accès à avoir lieu dans le segment du module courant et non pas dans un segment ne lui appartenant pas. Bien évidemment, la destination de l'accès sera alors complètement fantaisiste et l'exécution de l'accès mémoire aboutira vraisemblablement à une faute du module concerné, ce qui est considéré comme sans danger pour le système par les auteurs de cette technique. Cependant, elle nous paraît assez dangereuse, car la nouvelle adresse pourrait être celle d'une fonction dont l'exécution risquerait d'endommager le système. Avec beaucoup de malchance, on pourrait alors par exemple transformer un appel d'une fonction externe par un appel vers une fonction de formatage du disque dur.

Si on met de côté les objections formulées concernant l'innocuité de la technique du bac à sable, l'isolation par injection de code entraîne plusieurs remarques. Tout d'abord, les deux techniques présentées sont très peu flexibles. En effet, elles permettent juste d'implanter une isolation « tout ou rien », similaire à celle obtenue grâce à la segmentation matérielle. De plus, le surcoût à l'exécution est prohibitif dans le cas où le programme contient beaucoup d'accès mémoire intra-segment et peu d'accès inter-segments (dans l'exemple donné dans [WLAG93], on remplace 1 instruction par 5, ce qui évidemment est coûteux à l'exécution). Le gain en performance par rapport à l'isolation matérielle n'est réel que si le processus isolé réalise beaucoup d'accès inter-segments, ce qui n'est pas le cas de tous les programmes. Enfin, injecter du code augmente la taille des processus du système, ce qui peut être gênant s'il s'exécute sur un système embarqué où l'espace mémoire est limité.

2.2.2.3 Interprétation

L'interprétation d'un programme consiste à émuler le fonctionnement d'une machine réelle par le biais d'une machine virtuelle, c'est à dire un logiciel. La compilation d'un langage interprété génère le plus souvent un programme dans un langage intermédiaire, plus facilement interprétable que du langage machine. L'interprétation permet un contrôle total de l'exécution du code et l'implantation de nombreuses vérification dynamique. Si on prend l'exemple du langage Java, la machine virtuelle [LY99] effectue 4 types de vérifications lors qu'une classe est chargée. Tout d'abord, elle vérifie que le format du binaire représentant la classe est correct et marqué de la signature des classes Java (i.e. : les quatre premiers octets du fichier sont `0xCAFEBABE`). Ensuite, des vérifications sont effectuées sur l'arborescence des classes et la validité des prototypes de méthodes. Puis, des vérification complexes sont effectuées concernant la validité des types et variables manipulés par les méthodes de la classe. Enfin, les droits d'accès (e.g. : `private`, `protected`, ...) aux champs et méthodes de la classe sont vérifiés. Toutes ses vérifications permettent d'assurer qu'il est impossible de générer ou de modifier un fichier `.class` dont le code pourrait mettre en péril la stabilité de la machine virtuelle ou effectuer des accès interdits par exemple (bien qu'il soit tout à fait possible de générer un fichier `.class` sans passer par un compilateur Java, voire même d'insérer du code dans un fichier existant pour en modifier le comportement : ces vérifications assurent juste que ces éventuelles modifications seront sans danger pour le système). L'interprétation permet donc de mettre en place toutes les vérifications dynamiques nécessaires et elle garantit une isolation flexible des composants du système.

Le principal inconvénient de l'interprétation est bien évidemment le surcoût important à l'exécution des programmes. Un simple test comparatif pour un algorithme de tri à bulles se révèle environ deux fois plus lent en Java qu'en C (bien sûr, ce type de comparaison n'a aucune valeur générique car les performances dépendent fortement du type de programme). Des optimisations ont été recherchées pour augmenter les performances des programmes interprétés. La principale consiste à recompiler le programme interprété juste avant son exécution. Cette recompilation a lieu dans la machine virtuelle et reste transparente pour l'utilisateur. Le compilateur intégré à la machine virtuelle, que l'on appelle « compilateur juste à temps » (de l'anglais *Just In Time compiler*), transforme donc les programmes du langage intermédiaire vers du langage machine, ce qui permet ensuite à la machine virtuelle de les exécuter directement sans avoir à les interpréter.

Cette technique permet d'obtenir des gains de performances importants, et est maintenant intégrée dans la plupart des machines virtuelles, notamment les machines virtuelles Java. Elle engendre cependant des problèmes liés à la transformation du code, notamment l'augmentation de la taille de celui-ci (puisqu'en général le langage intermédiaire utilisé est beaucoup plus compact que du langage machine) et la perte de certaines méta-informations associées au programme (par exemple les types des variables utilisées dans le programme).

Un exemple d'un tel compilateur juste à temps optimisé pour compiler des programmes destiné à des systèmes embarqués pourvu d'une quantité de mémoire très limitée est détaillé dans [Rip99]. Les compilateurs juste à temps actuels ne permettent toutefois par encore d'obtenir des performances équivalentes à un programme directement compilé en langage machine. Enfin, un problème intrinsèque à l'interprétation est la nécessité de disposer d'une machine virtuelle dans l'environnement d'exécution des programmes, ce qui n'est en général pas envisageable pour les noyaux de systèmes d'exploitation destinés à des systèmes embarqués limités en espace mémoire et en puissance de calcul. Pour information, la machine virtuelle Java pour Linux d'IBM consomme environ 8 Mo de mémoire pour exécuter le simple programme de tri par bulles présenté plus haut.

2.2.2.4 Synthèse

L'isolation mémoire est un problème pour lequel de nombreuses solutions ont été proposées, sans qu'aucune d'entre elles ne se soit imposée. Les problèmes le plus souvent rencontrés sont le manque de flexibilité et le manque d'efficacité. Le choix d'une technique pour un système donné reste donc en général lié aux contraintes du programmeur système en terme de matériel supporté et du type de système qu'il construit. Ainsi, un programmeur construisant un système pour un processeur ne comprennant pas d'unité de gestion de la mémoire sera contraint d'utiliser un mécanisme d'isolation logicielle.

2.3 Gestion de la qualité de service

La gestion de la qualité de service dans un système d'exploitation consiste à répartir les différentes ressources du système entre les consommateurs selon une politique garantissant un résultat souhaité (notamment en terme de performances, de disponibilité, etc). On présente ci-dessous quelques techniques classiques de multiplexage des ressources principales d'un ordinateur.

2.3.1 Partage de la ressource processeur

Le partage du processeur consiste à entrelacer l'exécution des différents processus du système en fonction de la politique choisie. Cet entrelacement est réalisé par un logiciel appelé ordonnanceur (de l'anglais *scheduler*). Les principales notions et quelques algorithmes d'ordonnancement classiques sont décrites ci-dessous.

2.3.1.1 Notions importantes

Algorithmes sans réquisition Les algorithmes non préemptifs se caractérisent par le fait qu'un processus s'exécutant ne peut être interrompu et s'exécute donc jusqu'à terme

ou jusqu'à ce qu'il rende lui-même la main.

Algorithmes avec réquisition Les algorithmes préemptifs sont caractérisés par un temps d'exécution maximum, généralement appelé *quantum*, qui représente le temps pendant lequel le processus va s'exécuter s'il ne rend pas la main de lui même. Une fois ce quantum écoulé, le processus est interrompu par l'ordonnanceur qui le bloque et donne la main à un autre processus.

2.3.1.2 Algorithmes d'ordonnancement

Ordonnancement par ordre d'arrivée L'ordonnancement en file (de l'anglais *First-In, First-Out*) est le plus simple des algorithmes d'ordonnancement non préemptifs, et consiste à exécuter chaque processus dans l'ordre de création. Il est rarement utilisé à cause de son manque d'équité. En effet, un processus très long ne rendant pas la main avant son terme monopolisera le processeur, ce qui dans un système multi-utilisateurs n'est pas admissible.

Ordonnancement par temps d'exécution Cet algorithme se base sur l'idée qu'un processus pouvant arriver rapidement à son terme doit être exécuté avant un processus qui de toute façon sera long à terminer. Cela semble logique du point de vue de l'utilisateur, car un utilisateur exécutant une compilation d'un gros projet s'attend de toute façon à ce que cela prenne du temps, alors qu'il ne tolérera pas que la compilation d'un programme Hello World dure plusieurs minutes. En pratique, cet algorithme est très peu utilisé dans les systèmes classiques car il est souvent très difficile voir impossible de prévoir le temps d'exécution d'un processus, et on ne peut donc pas facilement classer les processus.

Ordonnancement circulaire L'ordonnancement circulaire, souvent appelé tourniquet (de l'anglais *round-robin*) est un des algorithmes préemptifs les plus couramment utilisés. Le principe est tout simplement de laisser s'exécuter un processus pendant son quantum de temps, et de le bloquer pour passer au suivant quand ce quantum est écoulé (ou avant si le processus se termine ou rend volontairement la main). Les processus ne sont pas triés et sont mis en attente dans un simple liste chaînée selon leur ordre de création. La seule difficulté dans la mise en œuvre de cet algorithme réside dans le choix du quantum. En effet, un quantum trop court aura pour conséquence de multiplier les changements de contexte qui sont coûteux, alors qu'un quantun trop long sera simplement inutile puisque le processus aura rendu la main de lui même ou se sera terminé avant la fin du quantum. En pratique, on trouve qu'un quantum entre 20 et 50 milli-secondes fait en général l'affaire et permet de garantir des temps de réponse corrects pour un utilisateur du système.

Ordonnancement à priorité Le principe de cet algorithme est d'affecter à chaque processus une priorité qui reflète l'importance du processus par rapport aux autres. Les processus sont alors ordonnés de façon à exécuter tout d'abord le processus ayant la plus haute priorité, puis le second, et ainsi de suite. Lorsque cet algorithme est implanté de façon préemptive, l'interruption d'un processus se fait en général au bout d'un quantum comme dans le cas de l'algorithme circulaire. Une autre technique consiste à décrémenter la priorité du processus au bout de chaque quantum, et de vérifier s'il n'existe pas alors de processus devenu plus prioritaire dans la file d'attente. En pratique cet algorithme permet principalement de différencier les processus systèmes tournant en arrière plan (les démons (de l'anglais *daemons*) d'un système Unix) des processus utilisateurs qui doivent bénéficier du maximum de temps d'exécution possible, alors que les démons peuvent être beaucoup moins prioritaires. Une variante très utilisée de cet algorithme consiste à regrouper les processus selon leur priorité, et à ordonnancer les processus à l'intérieur des groupes selon un algorithme circulaire.

Ordonnancement équitable Les algorithmes présentés ci-dessus ne prennent pas en compte l'utilisateur auquel appartient les processus ordonnancés. Ainsi, si un utilisateur exécute 9 processus en parallèle alors qu'un autre n'en lance qu'un, dans le cas d'un algorithme circulaire par exemple le premier utilisateur bénéficiera de 90% du temps d'exécution. Pour éviter cette iniquité entre les utilisateurs, l'ordonnancement équitable prend en compte les propriétaires des processus lorsqu'il ordonne la file d'attente, en entrelaçant les processus appartenant à des utilisateurs différents. Ainsi, dans l'exemple précédent, le processus du deuxième utilisateur sera exécuté à chaque changement de processus, de façon à ne jamais exécuter deux processus appartenant au premier utilisateur à la suite l'un de l'autre. On remarque que si cet algorithme est équitable du point de vue des utilisateurs, il ne l'est pas forcément du point de vue des processus, puisque dans notre exemple le processus du deuxième utilisateur est exécuté 10 fois plus souvent que ceux du premier utilisateur.

Ordonnancement temps-réel La problématique de l'ordonnancement dans un système temps-réel est très différente de celle dans un système classique. En effet, on doit prendre en compte les contraintes fixant la date limite d'exécution d'un processus lors de son ordonnancement. Ces contraintes peuvent être réparties en deux catégories, les contraintes dures qui doivent être impérativement respectées, et les contraintes légères qui peuvent être sacrifiées en cas de nécessité. Il existe de nombreux algorithmes d'ordonnancement temps-réel, notamment pour les applications multimédia qui nécessitent des temps de réponse courts pour garantir une bonne diffusion du son ou des images par exemple.

2.3.1.3 Synthèse

Il existe de nombreux algorithmes d'ordonnancement adaptés à différents types de systèmes et d'applications. Cependant, l'ordonnanceur est en général considéré comme un élément fondamental du noyau d'un système et ne peut être remplacé aisément. La politique d'ordonnancement est donc le plus souvent fixée une fois pour toutes lors de la compilation du système, ce qui manque clairement de flexibilité.

2.3.2 Partage de la ressource disque

Le partage de la ressource disque inclue deux problématiques bien différentes : le partage de l'espace disque d'une part, et le partage de l'accès au disque d'autre part (ce qu'on appelle parfois par analogie avec le réseau, le partage de la bande passante disque).

2.3.2.1 Partage de l'espace disque

Partager l'espace disponible sur un disque revient à définir des quotas d'occupation pour chaque utilisateur. La majorité des systèmes modernes permettent de spécifier la taille maximum occupée sur le disque par les données d'un utilisateur. La plupart des mécanismes de quotas implantés obligent à définir une taille fixe pour chaque utilisateur, mais des techniques récentes proposent la gestion de quotas élastiques [LNZO02] évoluant en fonction de la place disponible sur le disque.

2.3.2.2 Partage de l'accès au disque

Le partage de l'accès au disque est un problème beaucoup plus complexe et critique que le partage de l'espace disque. En effet, si de nos jours les disques de très grande taille sont devenus bon marché, il est toujours relativement aisé de dégrader leurs performances en saturant le système de requêtes de lecture et d'écriture au point de rendre l'exécution d'autres programmes si lente qu'ils en deviennent inutilisables. De nombreux algorithmes d'ordonnancement des requêtes ont été proposées, et nous détaillons les principaux ci-dessous.

Ordonnancement par ordre d'arrivée L'ordonnancement par ordre d'arrivée (de l'anglais *First Come, First Served*) consiste simplement à ne pas réorganiser les requêtes qui arrivent dans la file des requêtes et à les traiter séquentiellement.

Cet algorithme est peu utilisé à cause des très mauvaises performances qu'il engendre. En effet, si les requêtes émises par les différents processus sont destinées à des secteurs très éloignés sur le disque, la tête de lecture va passer plus de temps à se déplacer qu'à lire ou écrire, d'où une grande inefficacité.

Ordonnancement par temps de recherche croissant L'ordonnancement par temps de recherche croissant (de l'anglais *Shortest Seek Time First*) consiste à réorganiser les requêtes de façon à minimiser le déplacement de la tête de lecture. Pour cela, on exécute en premier les requêtes vers les secteurs les plus proches de la position actuelle de la tête de lecture (c'est à dire le dernier secteur accédé). La figure 2.3 donne un exemple d'une telle réorganisation.

Position initiale	: `S1`
Ordre initial	: `[P1S0, P1S3, P1S5, P2S2, P2S3, P2S4, P2S6]`
Réorganisation `SSTF`	: `[P1S0, P2S2, P1S3, P2S3, P2S4, P1S5, P2S6]`

FIG. 2.3 – Ordonnancement par l'algorithme *SSTF*

La position initiale est celle de la tête de lecture, ici sur le secteur 1. Une entrée de la forme `PxSy` signifie « requête émise par le processus `x` pour accéder au secteur `y` ».

Le principal inconvénient de cet algorithme est le risque de famine qu'il engendre. En effet, un processus malveillant pourrait émettre en boucle des requêtes pour des secteurs proches de la tête de lecture et un autre processus désirant accéder légitimement à un secteur éloigné ne serait alors jamais servi.

Algorithme « de l'ascenseur » L'algorithme d'ordonnancement connu sous le nom d'« algorithme de l'ascenseur » (aussi appelé algorithme *SCAN* dans la littérature anglaise) réorganise les requêtes de façon à provoquer un balayage bi-directionnel du disque par la tête de lecture. Ainsi, les requêtes vers les secteurs les plus proches de la tête de lecture et dont le traitement engendre un mouvement de la tête dans le sens courant seront traitées en priorité. Le sens courant change lorsqu'il n'y a plus de requêtes dont le secteur destination se trouve entre la position de la tête et l'extrémité du disque. La figure 2.4 donne un exemple d'application de cet algorithme.

Position initiale	: `S4`
Sens de lecture initial	: `croissant`
Ordre initial	: `[P1S0, P1S3, P1S5, P2S2, P2S3, P2S4, P2S6]`
Réorganisation SCAN	: `[P2S4, P1S5, P2S6, P1S3, P2S3, P2S2, P1S0]`

FIG. 2.4 – Ordonnancement par l'algorithme *SCAN*

Cet algorithme a l'avantage d'éviter la famine puisque toutes les requêtes seront traitées au bout d'un temps fini. Cependant, il est intrinsèquement inéquitable, puisque la tête de lecteur accédera plus fréquemment aux secteurs au centre du disque qu'à ceux aux extrémités. De plus, en pratique, on constate que le fait d'ignorer les requêtes vers des secteurs proches de la tête de lecture mais dont le traitement engendrerait un changement de sens alors que l'extrémité du disque n'est pas atteinte engendre une dégradation globale

des performances par rapport à l'algorithme précédent. En effet, il arrive fréquemment qu'un fichier soit fragmenté sur le disque, ce qui implique que les secteurs contenant ses données peuvent se trouver répartis de chaque côté de la tête de lecture. Dans ce cas, une lecture séquentielle du fichier sera très lente car les secteurs positionnés avant la tête par rapport au sens courant mettront du temps à être accédés.

Algorithme « de l'ascenseur à sens unique » Cet algorithme est une modification de l'algorithme précédent couramment appelé algorithme *C-SCAN* dans la littérature (de l'anglais *Cyclic-SCAN*). Le balayage du disque s'effectue uniquement dans le sens des numéros de secteurs croissants et la tête de lecture est ramenée directement au secteur 0 lorsqu'elle atteint la fin du disque[3], sans traiter de requêtes en revenant. La figure 2.5 donne un exemple de cette réorganisation.

Position initiale	: `S4`
Ordre initial	: `[P1S0, P1S3, P1S5, P2S2, P2S3, P2S4, P2S6]`
Réorganisation C-SCAN	: `[P2S4, P1S5, P2S6, P1S0, P2S2, P1S3, P2S3]`

FIG. 2.5 – Ordonnancement par l'algorithme *C-SCAN*

Cet algorithme ne souffre pas du même problème d'inéquité que l'algorithme précédent. La dégradation des performances en cas de fragmentation du fichier à lire est cependant toujours présente, car on continue d'ignorer les fragments situés avant la tête de lecture.

Algorithme « de l'ascenseur probabiliste » Cet algorithme, également connu sous le nom de *VSCAN(p)*, est un mélange de l'algorithme *SSTF* et de l'algorithme *SCAN*. Le paramètre `p` représente en effet la probabilité de continuer à traiter les requêtes dans le sens courant si la file contient des requêtes vers des secteurs plus proches mais nécessitant un changement de sens pour être traitées. Ainsi, *VSCAN(0.0)* est équivalent à *SSTF* et *VSCAN(1.0)* est équivalent à *SCAN*. Cela permet d'éviter le risque de famine de l'algorithme *SSTF* tout préservant les performances globales du système, en fonction du paramètre `p`. En pratique, on trouve qu'une probabilité de changement de sens de 20% est souvent le meilleur compromis, bien que cela dépende aussi de l'organisation des fichiers sur le disque.

[3]Un problème souvent ignoré lié à cet algorithme vient du fait que la méthode la plus évidente pour faire revenir la tête de lecture sur le secteur 0 consiste à lire ce secteur. Cela implique que le secteur 0 du disque sera lu très régulièrement, ce qui réduit d'autant plus sa durée de vie (les fabricants de disques garantissent un nombre maximal d'accès aux secteurs des disques, au-delà duquel la fiabilité n'est plus garantie). Le secteur 0 ayant un rôle essentiel pour le système, puisqu'il contient en général le code d'amorçage du système, une telle usure peut provoquer une panne fatale au système

2.3.2.3 Synthèse

De façon assez similaire au partage de la ressource processeur, il existe de nombreux algorithmes d'ordonnancement des requêtes d'accès au disque. Et de la même façon, la plupart des systèmes intègrent l'ordonnanceur de disque dans leur noyau sans laisser la possibilité de le changer dynamiquement. Ce manque de flexibilité est regrettable car il empêche d'adapter la politique d'ordonnancement en fonction des besoins des différents types d'applications pouvant s'exécuter dans le système. On arrive donc, d'un côté à des systèmes généralistes où l'algorithme d'ordonnancement des accès disques est choisi de façon à offrir des performances acceptables pour tous les types d'applications, et de l'autre, à des systèmes se consacrant à un type d'applications particulier dans lesquels l'algorithme est très performant pour les applications de ce type au détriment des autres types d'applications pour lequel il est beaucoup moins adapté.

2.3.3 Partage de la ressource réseau

Le partage de la ressource réseau consiste à assurer aux applications un accès au réseau offrant les performances les plus adaptées à leurs besoins spécifiques. Le partage de la bande passante entre les différentes applications d'un système est une part de cette problématique pour laquelle beaucoup de solutions ont été proposées. Un autre défi consiste à préserver les systèmes serveurs contre les attaques par déni de service [4] qui sont devenus très courantes ces dernières années. On traite ci-dessous de ces deux problématiques.

2.3.3.1 Partage de la bande passante réseau

Le partage de la bande passante entre les différentes applications nécessitant un accès au réseau est un problème classique notamment pour les systèmes offrant des services de contenus multimédia en ligne. De nombreux algorithmes ad-hoc ont été proposés pour les différents types d'applications (adaptation de vidéos, vidéo-conférences, ...), par exemple dans [BC00], [MP00] et [SS98]. Nous présentons ici un algorithme plus général utilisé notamment sur les réseaux ATM.

Algorithme min-max de partage de la bande passante Cet algorithme (en anglais *bandwidth min-max fairness*) se base sur la définition d'allocation réaliste de la bande passante (en anglais *feasible bandwidth allocation*) suivante : une allocation réaliste de la

[4] On utilise ici le terme « attaques par déni de service » (de l'anglais *Denial-of-Service attacks*) de façon abusive. En effet le terme « attaques par déni de service » englobe théoriquement n'importe quel type d'attaques visant à dégrader les performances d'un service offert, service qui n'est pas forcément lié au réseau (cette définition rejoint celle de B.W. Lampson dans [Lam71]). Cependant, le terme « attaques par déni de service » est en général employé de nos jours pour désigner les attaques lancées contre des serveurs sur Internet et on parle plutôt d'attaques contre la qualité de service dans le cas d'attaques contre des services systèmes.

bande passante est un vecteur $R = (r_0, r_1, ..., r_{n-1})$ tel que $\sum_{k=0}^{n} r_k \leq C$ où r_i est la bande passante allouée à l'application i, n le nombre d'applications et C la bande passante totale disponible. En supposant que les éléments de chaque vecteur R_x sont triés de telle façon que $\forall i \in [0; n-2]\, r_i \leq r_{i+1}$, on définit une relation d'ordre en deux allocations réalistes R^1 et R^2 par : $R^1 < R^2 \Leftrightarrow \exists\, j \; tel\; que \; \forall i \in [0; j[\; R_i^1 = R_i^2 \; et \; R_j^1 < R_j^2$. L'algorithme min-max renvoie alors la plus grande allocation réaliste selon cet ordre.

Cet algorithme a pour effet d'allouer le plus équitablement possible la bande passante entre les différentes applications en essayant de fournir à chacune d'entres elles la bande passante maximum dont elle a besoin. Il ne prend pas en compte les différents niveaux de criticité des applications et ne permet pas de définir des priorités, ce qui le rend peu flexible. De nombreux autres algorithmes ont été proposés pour prendre en compte des besoins spécifiques, notamment les contraintes temps-réel de certaines applications multimédia.

2.3.3.2 Attaques par déni de service

Les attaques visant les serveurs Internet (principalement les serveurs HTTP [Soc99] et FTP [PR85]) exploitent généralement des failles dans les protocoles des services offerts. La bataille engagée entre les pirates et les spécialistes de la sécurité tourne en général à l'avantage des pirates car il n'existe à l'heure actuelle (et il n'existera vraisemblablement jamais) aucun protocole parfaitement sûr. Cependant, des mesures peuvent être prises pour compliquer la tâche des pirates et les dissuader d'attaquer un système. On présente ci-dessous quelques exemples d'attaques par déni de service bien connues et les principaux moyens pour les contrer.

Le `ping` mortel Le `ping` mortel (de l'anglais *ping of death*) est une attaque exploitant une faille dans l'implantation du protocole IP [DAR81a] sur de nombreux systèmes. La spécification du protocole IP précise en effet qu'un paquet IP ne doit pas dépasser 64 Ko, ce que respectent la plupart des gestionnaires réseau lorsqu'ils émettent un tel paquet. Cependant, si la plupart des systèmes interdisent à un utilisateur de générer lui-même des paquets IP (ce qui nécessite de pouvoir créer des *sockets* brutes (de l'anglais *raw sockets*), ce qui est en général interdit aux utilisateurs), il existe un programme capable de générer de tels paquets IP et accessible à tous : le programme `ping`. Ce programme permet de tester si une machine est connectée au réseau et d'obtenir des informations sur la vitesse du réseau vers cette machine. Il se base pour cela sur le protocole ICMP [Pos81], qui fait partie intégrante du protocole IP et sert principalement à faire transiter des messages d'erreurs ou des informations sur la qualité du réseau entre différentes machines. Le programme `ping` est capable de générer des paquets IP pour encapsuler des requêtes appelées `echo` auxquelles la machine destination répondra par des requêtes `echo-reply` contenant les mêmes données afin de vérifier qu'il n'y a pas eu d'altération. La plupart des versions

du programme `ping` permettant de paramétrer la taille des données encapsulées dans les requêtes `echo`, il est facile de faire générer à `ping` des paquets IP de taille supérieure à 64 Ko. Comme la plupart des implantations de IP étaient initialement programmées en partant du principe qu'un paquet IP serait toujours plus petit que 64 Ko, une attaque de ce type engendrait en général des dépassements de capacité dans les tampons de réception des paquets IP lors de leur recomposition.

Cette attaque a connu un grand succès il y a quelques années, principalement à cause de sa facilité de mise en œuvre. Désormais la plupart des systèmes d'exploitation sont capables de gérer les paquets IP de taille supérieure à 64 Ko et de les rejeter, ce qui les immunise contre ce type d'attaque. Cependant, cette attaque est un bon exemple de ce qui peut arriver quand la spécification d'un protocole laisse à la charge du programmeur qui l'implante des choix critiques en matière de sécurité. En effet, s'il est bien précisé que la taille d'un paquet IP ne doit pas dépasser 64 Ko dans la spécification du protocole IP, il n'est pas précisé comment réagir au cas où on reçoit tout de même un paquet de taille supérieure, ce qui a eu pour conséquence que de nombreux programmeurs n'ont tout simplement pas pris en compte ce cas dans leur implantation du protocole IP.

L'inondation de requêtes Ce terme général englobe toutes les attaques basées sur l'idée consistant à saturer le serveur de requêtes afin de ralentir l'exécution des requêtes légitimes émises par d'autres utilisateurs. Ce type d'attaques peut aussi avoir comme effet l'arrêt du serveur en question si celui-ci est mal programmé ou si le protocole qu'il implante comporte des failles. C'est le cas notamment du protocole TCP [DAR81b] qui est vulnérable à un type d'attaque bien connu : les attaques par inondation de requêtes SYN (de l'anglais *SYN flooding attacks*).

Le protocole TCP permet l'établissement de connexions entre deux machines. Pour cela, il définit un mécanisme appelé protocole à trois phases (de l'anglais *three-way handshake*), détaillé par la figure 2.6.

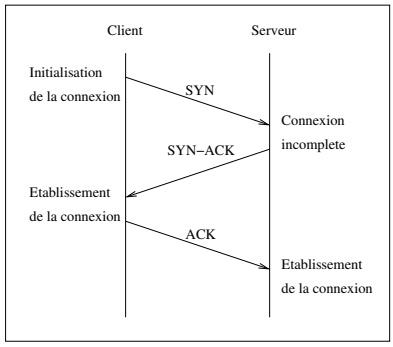

FIG. 2.6 – Protocole à trois phases

Tout d'abord le client envoie un paquet TCP dont un bit particulier, le bit SYN, a pour valeur un (par extension, on appellera ce paquet un paquet SYN). A la réception de ce paquet, le serveur renvoie un paquet dont les bits SYN et ACK ont pour valeur un (un paquet SYN-ACK). Le client, à la réception de ce paquet SYN-ACK, renvoie un paquet dont le bit ACK a pour valeur un (un paquet ACK) et la connexion est établie. La première phase permet donc de notifier au serveur que l'on désire établir une connexion avec lui, la deuxième permet au serveur de répondre au client qu'il est prêt à accepter une connexion, et la troisième sert à vérifier que le paquet SYN reçu correspondait bien à une demande de connexion (en effet, il est possible qu'un paquet SYN émis auparavant par le même client ait été retardé sur le réseau et arrive au serveur alors que le client ne veut plus établir de connexion).

Cependant, ce protocole engendre une vulnérabilité pour le serveur. En effet, lorsqu'il reçoit le paquet SYN, le serveur stocke ce paquet dans une file appelée file de stockage (de l'anglais *backlog queue*) afin de préserver les options et les paramètres de la connexion demandée par le client. Si le client ne répond pas par un paquet ACK dans un délai prédéfini (appelé en anglais *round trip time*), alors le serveur considère qu'il s'agissait d'une demande de connexion égarée et efface les informations stockées dans la file. Le problème vient du fait que le délai pendant lequel le serveur conserve les informations dans la file en attendant le paquet ACK du client est assez long, de l'ordre de 3 secondes en général (ce délai est en fait variable et dépend de la qualité du réseau mais il est en général surévalué afin d'éviter de rejeter des connexions légitimes). Un client malveillant peut donc émettre à haut débit une succession de paquets SYN en direction du serveur, sans jamais répondre aux paquets SYN-ACK renvoyés par celui-ci évidemment. Cela engendrera rapidement une saturation de la file de stockage, ce qui dans le meilleur des cas provoquera le rejet de demandes de connexion légitimes, et au pire un dépassement de la capacité de la file qui peut entraîner un arrêt du serveur.

Les *SYN cookies* : Les *SYN cookies* sont un mécanisme de défense bien connu contre les attaques par inondation de requêtes SYN. Le principe est simple : puisque le fait de stocker le paquet SYN dans la file de stockage rend le système vulnérable, alors on ne stocke pas les paquets SYN que l'on reçoit. Les options et paramètres de connexion transmis dans le paquet SYN sont en fait codées dans le paquet SYN-ACK renvoyé vers le client, qui les renverra par le biais du paquet ACK qui établit la connexion. Ces options encodées sont appelées *cookies* d'où le nom du mécanisme.

Cette solution est théoriquement intéressante car elle permet d'utiliser le réseau comme une sorte de file de stockage infinie, mais elle pose de nombreux problèmes pratiques. Tout d'abord, l'espace utilisable dans le paquet SYN-ACK n'est pas suffisant pour stocker toutes les informations transmises dans le paquet SYN et certaines optimisations de TCP ne sont

donc pas utilisables. De plus, ce mécanisme n'est pas compatible avec le protocole T/TCP [DAR81c] qui offre un support pour les transactions sur TCP.

Ensuite, ce mécanisme est beaucoup moins tolérant aux pannes que le protocole TCP. En effet, au cas où le paquet ACK renvoyé par le client se perd, TCP impose que le serveur renvoie le paquet SYN-ACK après un temps d'attente. Mais avec les *SYN cookies*, le serveur ne garde pas de trace de la demande de connexion et ne peut donc renvoyer le paquet SYN-ACK. En cas de perte du paquet ACK, le client se trouve donc dans un état incohérent par rapport au serveur, puisque le client pense que la connexion est établie (puisqu'il a reçu un paquet SYN-ACK et renvoyé un paquet ACK) alors que le serveur lui pense que la demande de connexion était une erreur et l'annule (puisqu'il n'a pas reçu de paquet ACK). Le client risque donc d'envoyer des données sur le réseau qui seront rejetées par le serveur puisque la connexion n'est pas établie, d'où des erreurs à gérer par le client.

Enfin, le fait d'utiliser le paquet ACK renvoyé par le client pour établir la connexion indépendamment des paquets SYN et SYN-ACK précédent rend le serveur vulnérable à une attaque par inondation de paquets ACK. L'encodage des options contenues dans le paquet SYN par le serveur lorsqu'il renvoie le paquet SYN-ACK au client est en fait un cryptage qui permet de s'assurer que le paquet ACK reçu en retour est bien une réponse à un paquet SYN-ACK lui-même envoyé en réponse d'un paquet SYN. Cependant, rien n'empêche un client malveillant de générer aléatoirement des *cookies* en espérant en trouver un qui correspondra à une connexion en attente. Les *cookies* étant codés sur 32 bits, ce type d'attaque est réaliste si on dispose d'un réseau rapide entre le client et le serveur. Les *SYN cookies* ne sont donc en général activés qu'en cas d'attaque afin de permettre au système de fonctionner en mode dégradé. La plupart des systèmes actuels proposent ce mécanisme mais il n'est jamais activé par défaut vu ses inconvénients.

2.3.3.3 Synthèse

Le besoin de flexibilité en ce qui concerne la protection du système contre les attaques par déni de service réseau peut paraître moins évident que dans le cas d'un algorithme d'ordonnancement des processus ou des requêtes disques. Cependant, l'exemple de *SYN cookies* nous montre que certains mécanismes ne peuvent pas être utilisés en permanence et ne doivent être activés que dans certaines conditions. De plus, on peut vouloir changer dynamiquement les paramètres de ces outils de protection comme par exemple le taux de remplissage de la file de stockage des paquets SYN à partir duquel on refuse les nouvelles demandes de connexion. La flexibilité est donc également souhaitable pour la gestion de la ressource réseau.

2.4 Conclusion

Ce chapitre présente quelques techniques de protection couramment employées dans les systèmes d'exploitation classiques et met en évidence le manque de flexibilité de la protection dans ces systèmes. En effet, une solution est en général choisie a priori pour chaque type de ressource à protéger et rien n'est prévu pour permettre au programmeur système d'en changer aisément. Dans un noyau comme Linux par exemple, l'isolation mémoire se base sur l'isolation matérielle fournie par le processeur, l'ordonnanceur de processus est basé sur un algorithme préemptif à priorité, l'ordonnanceur de requêtes disque fonctionne selon le principe de l'ascenseur à sens unique, et le gestionnaire réseau intègre d'office une protection contre les débordements de la file des requêtes et la gestion des *SYN cookies* en option. Tous ces mécanismes sont codés « en dur » dans le noyau et ne peuvent être remplacés sans avoir à modifier, massivement dans le cas du gestionnaire mémoire, d'autres parties du noyau.

De plus, les mécanismes eux-mêmes sont souvent peu flexibles car peu paramétrables. Par exemple, l'isolation mémoire matérielle permet uniquement d'effectuer un contrôle d'accès basé sur les segments source et destination de l'accès, alors qu'une isolation logicielle permet d'effectuer n'importe qu'elle vérification puisque celle-ci est programmée. Le manque de flexibilité est donc un problème majeur dans les mécanismes de protection couramment utilisés dans les systèmes classiques, et suggère l'utilisation d'approches de conception différentes pour les mécanismes destinés à être implantés dans des système flexibles. Nous présentons justement dans le chapitre suivant les principales architectures de systèmes ayant été conçues dans le but d'en améliorer la flexibilité.

Chapitre 3

Les architectures de systèmes

3.1 Introduction

La flexibilité des systèmes d'exploitation préoccupe leurs concepteurs depuis les tout premiers systèmes multi-utilisateurs. Dès qu'il a fallu concilier les besoins parfois contradictoires de plusieurs utilisateurs, on s'est en effet rendu compte qu'on ne pouvait se contenter d'un système figé dans lequel toutes les politiques d'administration ont été fixées à la compilation sans pouvoir être modifiées ensuite. On a donc cherché à améliorer la flexibilité des systèmes en réfléchissant à de nouvelles architectures plus adaptées à la modification dynamique du noyau. Cependant, nous montrons dans ce chapitre que cette amélioration de la flexibilité se fait le plus souvent au détriment d'une autre propriété du système, et notamment la sécurité qui nous intéresse ici.

Nous présentons dans ce chapitre différentes architectures de systèmes d'exploitation. Nous commençons par un historique des architectures de système et une rapide présentation du système Multics, resté célèbre pour avoir été un des systèmes les plus sûrs. Puis nous présentons différents prototypes de recherche présentant des caractéristiques intéressantes du point de vue de la protection et de la flexibilité. Enfin nous détaillons l'architecture THINK afin de mettre en évidence son intérêt dans la construction de systèmes flexibles.

3.2 Historique

L'architecture d'un système joue un rôle considérable dans son degré de flexibilité et de sécurité. Tout au long de l'évolution des systèmes d'exploitation, de nombreuses architectures ont été proposées, chacune prétendant bien évidemment corriger les déficiences de la précédente. On présente ci-dessous les quatre architectures les plus significatives, avant de présenter le système Multics qui a joué un rôle historique très important dans le domaine de la sécurité des systèmes d'exploitation.

3.2.1 Noyaux monolithiques

Un noyau monolithique intègre par défaut tous les services systèmes pouvant être utilisés par les applications. Il peut être vu comme une bibliothèque de bas niveau offrant aux applications une interface exhaustive de toutes les fonctionnalités du système. Multics [CV65] ainsi que le premier noyau Unix [RT74] développé en 1974 sont des exemples de noyaux monolithiques. Cette architecture entraîne en général de bonnes performances du système du fait de l'intégration de tous ses composants, ainsi qu'une bonne sécurité puisqu'on peut clairement définir la séparation entre la zone mémoire système et la zone mémoire applicative. Son inconvénient principal est le manque flagrant de flexibilité puisqu'il est impossible de changer un service système par un autre s'il ne convient pas. De plus, la structure monolithique du noyau engendre bien souvent un code peu modulaire et difficile à maintenir, donc peu fiable. Les anciennes versions des noyaux Linux [Aiv02] et Solaris [MM00] étaient aussi des noyaux monolithiques.

3.2.2 Noyaux extensibles

Les noyaux extensibles ont été développés pour répondre au manque de flexibilité des noyaux monolithiques. Un noyau extensible est un noyau monolithique offrant la possibilité de charger dynamiquement des modules, en général des pilotes de périphériques, qui sont alors considérés comme faisant partie intégrante du noyau. Cela permet de compiler statiquement un noyau minimal, et de l'étendre ensuite si nécessaire. Certains noyaux extensibles proposent de plus la possibilité de décharger les modules lorsqu'ils ne sont plus utilisés, comme les noyaux Linux et Solaris récents par exemple.

Le principal problème de cette architecture est son absence de sécurité. Dans le noyau Linux par exemple, un module chargé dynamiquement est exécuté dans le même espace d'adressage et avec les mêmes privilèges que le reste du noyau (un noyau extensible restant fondamentalement un noyau monolithique). Il est donc très facile d'introduire dans le système un programme malveillant ou bogué capable de provoquer des dégâts importants. Plusieurs techniques peuvent cependant être mises en œuvre pour sécuriser l'insertion de modules dynamiquement dans le noyau, notamment en associant au code une preuve de son innocuité ou une signature électronique garantissant son origine.

3.2.3 Micro-noyaux

L'architecture micro-noyau est un pas de plus vers la miniaturisation du noyau d'un système d'exploitation (bien qu'historiquement elle ait été proposée avant l'architecture de noyaux extensibles). Le principe est de diviser le noyau entre d'un côté les services dit de base et considérés comme indispensables (par exemple le gestionnaire mémoire, l'ordonnanceur, ...) et de l'autre les services supplémentaires tels que des pilotes de périphériques

ou des gestionnaires de systèmes de fichier par exemple. Le noyau d'une part, et tous les services supplémentaires de l'autre sont isolés matériellement, selon un modèle proche de celui utilisé dans la gestion des processus sous UNIX. Cela permet de démarrer et d'arrêter les services supplémentaires selon les besoins du système. A la différence des noyaux extensibles, ces services restent cependant toujours en mémoire ce qui d'un côté permet d'accélérer leur mise en service car il n'est pas nécessaire de les charger depuis le disque dur (ou le réseau) mais de l'autre bien évidemment consomme de l'espace mémoire inutilement lorsqu'ils sont arrêtés.

Un problème important de cette isolation matérielle est le coût des appels inter-services, qui s'imposent ici à chaque appel à un service supplémentaire. Les premiers micro-noyaux tels que Amoeba [MVRT+90], Chorus [Gie90] ou Mach [ABB+86] souffraient particulièrement de ce problème de performances. Les micro-noyaux plus récents, comme L4 [Lie96] ou QNX [Hil92] par exemple ont eux été fortement optimisés afin de fournir des communications inter-services acceptables. Cependant, le principal inconvénient de l'architecture micro-noyau réside dans la définition des services de base. En effet, classer un service système comme un service de base ou un service supplémentaire est une définition subjective qui dépend souvent de l'utilisation qu'on compte faire du système. Ainsi, le gestionnaire mémoire est le plus souvent considéré comme un service de base du système, alors qu'un programmeur peut avoir besoin de le remplacer par son propre gestionnaire (par exemple pour remplacer un gestionnaire de mémoire segmentée par un gestionnaire de mémoire plate). L'architecture micro-noyau souffre donc d'un manque de flexibilité moins important que les noyaux monolithiques mais bien réel.

3.2.4 Nano-noyaux

L'architecture nano-noyau représente l'étape ultime dans la course vers la miniaturisation du noyau. Un nano-noyau ne fournit aucune abstraction et ne fait que réifier le matériel sous-jacent sous forme d'interfaces offertes aux services systèmes et aux applications. Le but est bien évidemment de permettre au programmeur de construire un système à la carte en ne lui imposant aucune abstraction par défaut. Cela laisse bien sûr à sa propre charge le développement de toutes les abstractions dont il aura besoin, ce qui représente un travail très important (l'exo-noyau que l'on détaille ci-après propose une solution à ce problème sous la forme d'une bibliothèque d'abstractions optionnelles). Un autre intérêt de cette architecture est de faciliter la portabilité d'un système d'une architecture matérielle à une autre. Par exemple, le noyau de Windows XP [BSS01] est basée sur un nano-noyau appelé « couche d'abstraction matérielle » (de l'anglais *Hardware Abstraction Layer*) qui propose une vue virtuelle du processeur sous-jacent grâce à une interface générique (cette architecture est un héritage du temps où Windows NT fonctionnait sur les plate-formes MIPS, Alpha et PowerPC, ce qui n'est plus le cas actuellement). Ainsi, la plupart des

services systèmes de base restent indépendant de la plate-forme sous-jacente.

3.2.5 Multics

Multics [CV65] a encore de nos jours la réputation d'avoir été un système particulièrement sûr. Ce système est l'ancêtre des systèmes modernes et a notamment fortement inspiré le système Unix [RT74]. Nous présentons ici les caractéristiques principales des mécanismes de protection mis en œuvre dans Multics afin de montrer que la problématique de la protection d'un système a somme toute bien peu changé depuis les premiers systèmes multi-utilisateurs, bien que l'environnement d'exécution de ces systèmes a lui beaucoup évolué puisque même les machines appartenant à des particuliers sont désormais reliées à Internet.

La protection dans Multics peut être découpée en deux niveaux [Sal74]. Au niveau de l'isolation mémoire, Multics s'appuie sur le mécanisme de segmentation fourni par le processeur sous-jacent (Multics a été implanté sur une machine conçue spécialement pour lui, le Honeywell 6180). Les segments, en plus de comprendre les informations de contrôle d'accès classiques, sont regroupés en anneaux de protection [Gra67]. Ces anneaux sont ordonnés en fonction du niveau de privilège que l'on souhaite accorder aux segments les composant. Les anneaux sont numérotés de 0 à 7, et la règle régissant les appels inter-anneaux consiste simplement à n'autoriser que les appels provenant d'un anneau de niveau inférieur à l'anneau contenant le segment de destination. L'anneau de niveau 0 contient donc typiquement le noyau du système et l'anneau de niveau 7 les applications des utilisateurs. Les appels inter-anneaux sont effectués via un mécanisme de portails (de l'anglais *gates*), parfois aussi appelés guichets, qui permettent d'effectuer des contrôles supplémentaires lors des appels. Ces mécanismes d'anneaux et de portails sont très similaires à ceux mises en œuvre dans le processeur Pentium d'Intel [Int97].

Au niveau du contrôle d'accès applicatif, Multics se base sur des listes de contrôle d'accès semblables à celles présentées par Lampson dans [Lam71]. Cependant, la gestion de ces listes est rendue très flexible par l'introduction d'identificateurs composés de trois niveaux. Le premier niveau permet de spécifier un utilisateur individuel par son nom. Le deuxième niveau permet de définir des groupes d'utilisateurs ayant besoin de partager des données ou des programmes. Le troisième niveau permet de définir des compartiments dans lesquels les utilisateurs pourront eux-même choisir d'exécuter tel ou tel programme. Par exemple, si l'utilisateur `rippert` fait partie du groupe `sardes`, il peut décider de définir deux compartiments `méfiance` et `confiance`. Les listes de contrôles d'accès de cet utilisateur peuvent alors être définie de façon à autoriser l'accès à certains fichiers précieux lorsque l'identifiant de l'utilisateur est `rippert.sardes.confiance` et à interdire l'accès à ces fichiers lorsque l'identifiant est `rippert.sardes.méfiance` (et aussi bien sûr lorsque l'identifiant ne commence pas par `rippert.sardes` de manière à interdire l'accès à ces

fichiers aux autres utilisateurs du système). Ainsi, lorsque l'utilisateur exécute un programme dont il estime qu'il peut poser un risque (par exemple une application téléchargée dont l'origine est douteuse), il peut s'identifier en tant que `rippert.sardes.méfiance` pour exécuter cette application dans un environnement aux droits restreints.

De façon générale, le système Multics a été développé de manière à choisir par défaut la solution la plus sûre pour chaque prise de décision. Par exemple, les listes de contrôle d'accès sont basées sur une politique fermée qui refuse par défaut l'accès lorsque l'identifiant de l'utilisateur n'est pas explicitement cité dans la liste de contrôle d'accès. Ce souci constant de la sécurité a fait de Multics un des systèmes les plus sûrs ayant existé. Malheureusement les concepteurs des systèmes lui ayant succédé, comme Unix par exemple, ont cherché à simplifier la gestion de la protection en considérant qu'un tel niveau de sécurité n'était pas requis pour des systèmes grand public. Il peut être amusant de noter qu'à ce sujet une évaluation de Multics réalisée par l'armée Américaine [KS74] a conclu que Multics, bien que beaucoup plus sûr que ces concurrents de l'époque, ne l'était pas assez pour être utilisé dans un contexte militaire et était donc destiné exclusivement à des utilisations civiles. Les auteurs de cette évaluation ont d'ailleurs récemment publié une étude [KS02] montrant que la plupart des failles de sécurité couramment rencontrées dans les systèmes actuels n'existeraient pas si les mécanismes de protection de Multics avaient été mis en œuvre dans ces systèmes.

3.3 Prototypes de recherche

On présente dans cette section quelques prototypes de recherche récents proposant des architectures de système intéressantes du point de vue de la protection et de la flexibilité. Il s'agit de fournir une vue d'ensemble du domaine via quelques exemples intéressants. Nous présentons leurs points forts et mettons en évidence leurs faiblesses sur le plan de la flexibilité et de la sécurité.

3.3.1 Noyaux extensibles

3.3.1.1 DEIMOS

DEIMOS [CC98] est un système extensible dont la particularité est de ne pas intégrer de noyau à proprement parler. En effet, tous les services, y compris le noyau, sont organisés sous forme de modules pouvant être chargés, configurés, et déchargés à la demande par un composant gestionnaire de configuration. Cette architecture donne une grande flexibilité au système puisque les applications peuvent configurer leur environnement d'exécution selon leurs besoins, sans qu'aucune abstraction ne leur soit imposée. De plus, le composant gestionnaire de configuration permet la reconfiguration dynamique des liaisons entre les

modules ce qui permet la modification à l'exécution de l'environnement d'exécution d'une application. Cependant, cette centralisation de la gestion des configurations dans un seul composant gestionnaire pose un problème de passage à l'échelle, car le gestionnaire doit maintenir en permanence un graphe décrivant l'état de configuration du système, graphe qui peut devenir très volumineux pour un système distribué sur beaucoup de nœuds.

3.3.1.2 Vino

Vino [SESS96] est un noyau extensible fournissant un mécanisme de sécurisation des modules supplémentaires chargés dynamiquement. Les modules sont chargés dans des espaces isolés grâce à un mécanisme d'isolation logicielle, et tous les appels vers ses modules sont encapsulés dans des transactions, afin de permettre leur annulation en cas de panne ou d'attaque. Ce mécanisme répond aux problèmes de sécurité caractéristiques des noyaux extensibles, mais au prix d'un surcoût à l'exécution prohibitif. En effet, le fait d'encapsuler chaque appel à une fonction du module dans une transaction est très coûteux et risque de pénaliser grandement les performances du système, par exemple si le module en question est un pilote de périphérique.

3.3.2 Micro-noyaux

3.3.2.1 DTOS

Le Distributed Trusted Operating System [OFSS96] propose une architecture de systèmes basée sur le micro-noyau Mach. Le but de cette architecture est de permettre la définition de politiques de sécurité quelconques. Le noyau lui-même reste indépendant de la politique choisie, qui est gérée par un composant externe spécialisé dans la gestion des politiques. Le projet DTOS assure la validité de ses politiques de sécurité par le biais de démonstration formelles.

Cette architecture est particulièrement intéressante dans la volonté affichée de séparer la gestion de la politique de sécurité de sa mise en œuvre. En effet le noyau reste complètement indépendant de la politique choisie par le biais d'outils et de points d'accès ajoutés au noyau Mach original afin de permettre la communication avec le serveur de sécurité externe au noyau. Du point de vue de la gestion flexible de la sécurité, cette architecture se révèle particulièrement intéressante. Par contre, elle ne propose pas de véritable flexibilité lors de la construction du noyau puisqu'on se retrouve sur ce point dans le contexte d'un micro-noyau imposant des abstractions jugées indispensables.

Il est intéressant de noter que cette idée de séparer la gestion de la politique des outils permettant sa mise en œuvre n'est pas nouvelle. Le système Hydra [WLP75] [CJ75] [LCC+75] proposait déjà une telle séparation afin de permettre aux utilisateurs et aux programmeurs d'applications de paramétrer la politique de sécurité du système en fonction

de leur besoin. Cependant, ce système étant intimement lié à l'architecture matérielle sur laquelle il s'exécutait (le multi-processeur C.mmp de l'université de Carnegie-Mellon), ces résultats n'ont pas été exploités dans d'autres environnements.

3.3.2.2 Kea

Le système Kea [VH96] est un micro-noyau intégrant un mécanisme de liaison entre services proche des liaisons inter-composant du modèle ODP [ODP]. Ce mécanisme supporte la reconfiguration dynamique d'une liaison, ce qui permet de changer dynamiquement le gestionnaire d'une ressource pour faire évoluer sa politique de gestion par exemple. Cependant, aucun mécanisme n'est prévu pour permettre aux applications d'influencer ces politiques de gestion en fonction de leurs besoins propres. De plus, Kea souffre du problème de performances commun à tout les micro-noyaux concernant les communications inter-processus qui doivent s'effectuer via des mécanismes coûteux contournant l'isolation des processus.

3.3.2.3 Pebble

Peeble [GSB+99] est un micro-noyau moderne axé sur la flexibilité, la sécurité, et les performances. Le micro-noyau lui même est limité au strict minimum et sert principalement à générer dynamiquement le code nécessaire aux changements de domaines de protection. Cette génération de code est une des particularité du système Pebble et permet d'optimiser les changements de contexte entre les différents services hors-noyau du système qui sont isolés matériellement les uns des autres. Cela permet au système d'obtenir des performances supérieures à celles obtenues en général par les micro-noyaux, bien que les appels inter-domaines de protection restent tout de même coûteux. Grâce à son micro-noyau limité, le système Pebble se rapproche de l'exo-noyau en terme de flexibilité. Cependant, on notera que tout le système de protection est basé sur l'isolation matérielle fournie par le processeur, ce qui rend le système inutilisable pour des architectures matérielles n'intégrant pas de tels mécanismes d'isolation.

3.3.2.4 Spin

Le micro-noyau extensible SPIN [BCE+94] [BSP+95] permet le chargement dynamique de code dans le noyau en fonction des besoins des applications. Cela permet de limiter le noyau de base au strict minimum et aux applications de paramétrer le système en chargeant les abstractions dont elles ont besoin. Cette approche est particulièrement risquée puisqu'un module malveillant ou même simplement bogué pourrait être chargé et exécuté en mode superviseur. Pour parer ce risque, le système SPIN utilise les propriétés de sûreté du langage Modula-3. Ce type de sécurité basé sur le langage implique en général une

dégradation non-négligeable des performances des composants. Cette dégradation des performances provient en général de mécanismes de vérifications dynamiques, comme celle consistant à s'assurer qu'un accès à un tableau ne s'effectue pas en dehors des limites de ce tableau par exemple. Ce type de vérification est bien entendu très coûteux car il implique d'ajouter deux comparaisons à chaque accès mémoire au tableau mais on notera que ces vérifications peuvent souvent être désactivées (cela dépend du compilateur). Cependant, la décision d'activer ou non telle ou telle vérification est forcément prise au moment de la compilation du système, ce qui ne permet pas ensuite de modifier dynamiquement les décisions prises si elles ne sont plus adaptées.

Concernant la gestion de ressources, le système SPIN propose une architecture à deux niveaux. Le niveau bas intègre des allocateurs systèmes chargés de gérer les ressources de bas niveau comme les pages mémoire, le processeur ou la bande passante réseau par exemple. Le niveau haut est composé d'allocateurs applicatifs qui gèrent des ensembles de ressources « achetées » par les applications aux allocateurs systèmes. Cette architecture permet aux applications de paramétrer la gestion des ressources dans les allocateurs applicatifs en fonction de leur besoin. Elle est cependant insuffisante car elle ne permet pas de paramétrer la gestion des ressources de bas niveau, dont la gestion est figée dans le code des gestionnaires.

3.3.3 Nano-noyaux

3.3.3.1 L'exo-noyau

L'exo-noyau [EKO95] se propose « d'exterminer toutes les abstractions systèmes » [EK95] qui imposent des contraintes aux programmeurs d'applications. En effet, dans un système comme Linux par exemple, il n'est pas possible de remplacer le gestionnaire de mémoire ou l'ordonnanceur si ceux-ci ne conviennent pas à une application particulière (par exemple, une application temps-réel nécessite un ordonnanceur approprié). L'exo-noyau résout ce problème en permettant la gestion des ressources du système au niveau applicatif. Le noyau de l'exo-noyau est en fait un nano-noyau qui ne fait qu'exporter les ressources matérielles de façon sécurisée à travers une interface accessible aux applications et n'ajoute aucune abstraction. Les bibliothèques du système implantées au dessus de ce nano-noyau peuvent alors utiliser cette interface pour construire les abstractions voulues. Cette séparation de la protection des ressources de leur gestion permet aux applications de paramétrer les abstractions systèmes en les modifiant ou même en les remplaçant complètement. Le prototypage de cette architecture a montré qu'elle se révélait très performante et qu'elle permettait un niveau de flexibilité bien supérieur à celui des systèmes monolithiques classiques.

Cette architecture est très intéressante dans sa volonté affichée de laisser la plus grande

liberté possible au programmeur pour construire un système à la carte. On notera cependant que certaines abstractions comme la notion de processus par exemple restent présentes dans l'exo-noyau lui-même, ce qui peut être contraignant pour le programmeur d'application. De plus, le modèle de programmation associé à cette architecture exo-noyau n'est pas clairement défini et ne met notamment pas en évidence la notion de composant pour les services inclus dans les bibliothèques d'abstraction.

3.3.3.2 Nemesis

Le système Nemesis [LMB+96] est spécialisé pour les applications nécessitant un contrôle fin de la qualité de service, comme les applications multimédia par exemple. Ce contrôle est effectué par un composant gestionnaire de la qualité de service, qui alloue à chaque application un temps de traitement équitable pour ses requêtes et qui permet un traitement prioritaire des requêtes nécessitant un temps de réponse très court. De plus, ce gestionnaire permet de définir des politiques d'allocation de ressources à long terme, afin d'anticiper les demandes des applications. Cette gestion centralisée de la qualité de service pose encore une fois le problème du passage à l'échelle, puisque ce gestionnaire risque d'être rapidement saturé dans un environnement distribué ou intégrant beaucoup d'applications posant des contraintes temps-réel. De plus, la politique d'allocation des ressources est basée sur un algorithme de résolution de contraintes qui ne peut être paramétré par les applications, ce qui entraîne un certain manque de flexibilité dans l'allocation des ressources.

3.3.4 Architectures pour la construction de systèmes

3.3.4.1 MMLite

MMLite [HF98] est une architecture de système modulaires développée par Microsoft. Cette architecture est basée sur le modèle à composants COM [Bro95] et permet de construire des systèmes à la carte. Elle intègre de plus un mécanisme permettant le remplacement dynamique d'un composant par un composant « équivalent » (c'est à dire un composant exportant la même interface). Cependant ce mécanisme est incomplet car il ne fourni pas d'outils pour s'assurer de la validité du remplacement. En effet, le remplacement dynamique d'un composant par un autre pose plusieurs problèmes non triviaux. Tout d'abord, il faut être capable de convertir l'état courant du composant en fonction des champs du nouveau composant, ce qui peut être particulièrement coûteux et délicat si l'état du dit composant est formé de graphes d'objets complexes. De plus, il n'est pas forcément possible d'arrêter l'exécution d'un composant à n'importe quel moment et on doit donc définir des points de synchronisation où le changement est possible. Enfin, au cas où le changement s'avèrerait impossible ou provoquerait des erreurs, on doit être en mesure

de restaurer l'ancien composant dans son état au moment du changement, ce qui implique d'utiliser un mécanisme transactionnel doté d'une opération d'annulation (*rollback*). L'architecture MMLite laisse toutes ses opérations complexes à la charge du programmeur. Un autre inconvénient de l'architecture MMLite est qu'elle est basée sur le modèle COM qui un modèle propriétaire dont seul Microsoft contrôle les évolutions.

3.3.4.2 OSKit

OSKit [FBB+97] est une architecture de systèmes d'exploitation permettant de composer des systèmes à la carte. A partir de code récupéré dans des systèmes d'exploitation existants comme Linux par exemple, il est possible de construire un noyau adapté aux besoins des applications. Cependant, l'assemblage des composants, réalisé selon un modèle d'architecture nommé Knit [RFS+00], est purement statique et ne prévoit aucun modèle de reconfiguration dynamique du système. De plus, certains composants de OSKit, comme le gestionnaire mémoire par exemple, sont très difficilement remplaçables car d'autres composants les utilisent de façon dépendante de leur implantation. L'architecture OSKit est donc une plate-forme intéressante pour la construction rapide de systèmes pour des machines classiques mais on peut regretter que l'effort de modularisation et d'indépendance des composants n'ait pas été poussé jusqu'au bout.

3.3.5 Systèmes dédiés

3.3.5.1 Inferno

Inferno [DPP+97] est un système d'exploitation commercial développé initialement par les laboratoires Bell destiné particulièrement aux systèmes embarqués de faible puissance. Les ressources sont représentées dans Inferno sous forme d'un système de fichiers hiérarchique. Cela permet aux applications de construire une vue sur mesure du système contenant uniquement les ressources pertinentes pour elles. Cette approche fournit de plus une unification de l'accès aux ressources du système, aussi bien locales que distantes. Cependant, la construction des vues des ressources est essentiellement statique et ne permet pas d'évolution dynamique de la gestion des ressources du système.

3.3.5.2 Scout/Escort

L'architecture Escort [SP99] permet de combiner le comptage des ressources utilisées dans un système (en l'occurrence ici le système Scout [MP96], spécialisé pour les services réseau) avec la définition de multiple domaines de sécurité. Le comptage des ressources se fait grâce à la définition de la notion de chemin d'entrée/sortie, qui représente la succession des modules (i.e : pilote de périphériques, serveurs, etc) traversés par les données. Par exemple, une requête HTTP génèrera vraisemblablement un chemin composé d'un

client web, d'un gestionnaire de mémoire, d'un gestionnaire réseau et d'un serveur HTTP, puisque tous ces modules sont impliqués dans l'émission et le traitement de la requête. Cela permet une facturation précise de la consommation de ressource à chaque module impliqué dans l'opération. Du point de vue de l'isolation des modules, le système Scout utilise les capacités d'isolation du processeur sous-jacent. Cette architecture est intéressante en ce qui concerne le mécanisme d'identification des consommateurs car la plupart des mécanismes de comptage des ressources ne prennent en compte que le client final et pas les intermédiaires, ce qui manque évidemment de précision. Par contre, le mécanisme d'isolation est proche de ce qu'on peut trouver dans un micro-noyau, avec toutes les conséquences en matière de performance des appels inter-domaines que cela implique.

3.4 L'architecture de systèmes flexibles Think

Nous présentons ici brièvement l'architecture de systèmes flexibles THINK. Une description plus détaillée peut être trouvée dans [Fas01] et [FSLM02].

3.4.1 Présentation

L'architecture de systèmes flexibles THINK[1] a pour but de fournir aux développeurs de systèmes un modèle de programmation et un ensemble d'outils pour faciliter la construction du système. Cette architecture est structurée autour des trois entités présentées ci-dessous.

3.4.1.1 Le nano-noyau

Le noyau minimal de l'architecture THINK (c'est à dire la portion de code que l'on doit obligatoirement retrouver dans tout système construit grâce à l'architecture THINK) ne doit en aucun cas imposer d'abstractions système afin de préserver la liberté du programmeur système. Son rôle est donc uniquement de réifier le matériel sous-jacent en fournissant les interfaces appropriées. Il s'agit donc d'un nano-noyau comme défini précédemment.

3.4.1.2 La bibliothèque de services systèmes

L'architecture THINK est plus qu'un nano-noyau car elle propose au programmeur système un ensemble de services systèmes programmés sous la forme de composants THINK mis à disposition dans une bibliothèque. Chaque service est programmé sous la forme d'un composant indépendant de l'implantation des autres. Cela signifie qu'un composant peut dépendre d'un autre (e.g. un composant système de fichier dépendra vrai-

[1] THINK est l'acronyme de *THink Is Not a Kernel*, ce qu'on pourrait traduire par « Think n'est pas (qu')un noyau »

semblablement d'un composant pilote de disque dur), mais uniquement via son interface. On peut donc remplacer un composant par un autre sans compromettre le fonctionnement de ceux dont il dépend, tant que le nouveau composant implante la même interface que l'ancien. On obtient ainsi une modularité complète puisque aucun composant n'est obligatoire.

3.4.1.3 Le canevas logiciel

En plus de fournir une bibliothèque de composants systèmes, l'architecture THINK propose un modèle de programmation facilitant l'écriture de composants systèmes. Ce modèle, inspiré par celui de l'ODP, est matérialisé par un canevas logiciel dans l'architecture THINK.

Principales notions du canevas logiciel

Composant : un composant est une entité encapsulant un état (caractérisé par l'ensemble des valeurs de ses champs) et un comportement (définit par l'ensemble des méthodes du composant). Cette définition est directement tirée de la définition ODP d'un objet.

Interface : une interface est une abstraction d'un sous-ensemble du comportement d'un composant. On parle ici de sous-ensemble car un composant peut implanter plusieurs interfaces.

Nom : un nom dans THINK est un moyen de désigner de façon déterministe une interface. Tous les noms dans THINK sont relatifs à un contexte de nommage.

Contexte de nommage : un contexte de nommage est un ensemble de noms valides, c'est à dire associé chacun à une et une seule interface. Un nom peut être associé à plusieurs interfaces différentes dans différents contextes de nommage et une interface peut avoir plusieurs noms dans des contextes de nommage différents.

Liaison : une liaison est un canal de communication entre deux composants. Un exemple typique de liaison, appelée liaison langage, est l'association effectuée par un compilateur entre le nom symbolique d'une variable locale et l'espace mémoire qui lui est réservé dans la pile.

Usine à liaisons : une usine à liaisons est un composant chargé de créer des liaisons entre composants.

Domaine : un domaine est un ensemble de composants groupés selon une propriété commune. Une telle propriété peut être par exemple une vulnérabilité à un même type de pannes (domaine de faute) ou la résistance à un type particulier d'attaques (domaine de sécurité).

Courtier : un courtier est un composant gérant l'association entre le nom d'un composant (tel que défini plus haut) et un ou plusieurs noms symboliques décrivant les propriétés du composant. Un tel service permet non seulement de faciliter la tâche du programmeur en offrant un moyen de désigner un composant de façon plus parlante qu'un nom THINK, mais aussi de classer les composants en fonction des services qu'ils offrent, afin par exemple de permettre des recherches parmi tous les composants offrant le service requis.

Implantation du canevas logiciel L'interface `Top` (figure 3.1) représente le plus grand élément du treillis des types de THINK. Elle concrétise la notion de type générique qui est souvent utilisée par les méthodes acceptant un argument de types multiples.

```
interface Top {
}
```

FIG. 3.1 – L'interface `Top`

L'interface `Name` (figure 3.2) représente les noms dans THINK. La méthode `getDefaultNC` permet d'obtenir le contexte de nommage dans lequel est définit le nom en question. La méthode `toByte` renvoie une version sérialisée (sous forme d'une chaîne de caractères) du nom.

```
interface Name {
  NamingContext getDefaultNC() ;
  String toByte() ;
}
```

FIG. 3.2 – L'interface `Name`

L'interface `NamingContext` (figure 3.3) représente le concept de contexte de nommage. La méthode `byteToName` permet de désérialiser un nom donné sous forme de chaîne de caractères. La méthode `export` permet d'exporter une interface, ce qui a pour effet de créer une liaison entre cette interface et le nom associé créé par la méthode `export`.

```
interface NamingContext {
  Name byteToName(String string) ;
  Name export(Top itf) ;
}
```

FIG. 3.3 – L'interface `NamingContext`

L'interface `BindingFactory` (figure 3.4) représente les usines à liaison. La méthode `bind` permet de créer une liaison entre le composant appelant la méthode et celui dont le

nom est donné en argument. L'interface renvoyée est celle du composant dont le nom est donné en argument.

```
interface BindingFactory {
  Top bind(Name name);
}
```

FIG. 3.4 – L'interface `BindingFactory`

L'interface `Trader` (figure 3.5) représente un courtier dans THINK. La méthode `register` permet de créer l'association entre le nom d'un composant (1er argument) et un nom symbolique (2e argument). La méthode `lookup` renvoie le nom associé au nom symbolique donné en argument. Le 2e paramètre de cette méthode permet de préciser quel nom doit être renvoyé au cas où plusieurs seraient enregistrés sous le même nom symbolique (il s'agit d'un index). La méthode `lookupfirst` est un raccourci pour l'appel `lookup(symbName, 1)`.

```
interface Trader {
  void register(Name name, String symbName);
  Name lookup(String symbName, int i);
  Name lookupfirst(String symbName);
}
```

FIG. 3.5 – L'interface `Trader`

Exemple d'utilisation du canevas logiciel La figure 3.6 illustre comment un composant serveur exporte une interface et s'enregistre auprès du courtier.

```
...
ServCompItf servCompItf;                                    // 1)
Name servCompName = namingContext.export(servCompItf); // 2)
trader.register(servCompName, "myServComponent");      // 3)
...
```

FIG. 3.6 – Exemple d'utilisation du canevas logiciel : exportation et enregistrement

1. La variable `servCompItf` est l'interface du composant serveur dont le type a été définit par le programmeur de ce composant.
2. La variable `namingContext` est le contexte de nommage dans lequel on désire créer le nom du composant serveur.
3. La variable `trader` est le courtier dans lequel on désire enregistrer le service fournit par le composant serveur.

La figure 3.7 détaille la procédure permettant à un composant client d'appeler la méthode `alloc` du composant serveur précédent.

```
...
ServCompItf servCompItf ;                                    // 1)
Name servCompName = trader.lookupfirst("myServComponent") ;  // 2)
servCompItf = bindingFactory.bind(servCompName) ;            // 3)
servCompItf.alloc(1234) ;                                    // 4)
...
```

FIG. 3.7 – Exemple d'utilisation du canevas logiciel : appel de méthode

1. La variable `servCompItf` est l'interface du composant serveur.
2. La variable `servCompName` est le nom de l'interface du composant serveur. La variable `trader` est le courtier dans lequel a été enregistré le service fournit par le composant serveur.
3. La variable `bindingFactory` est l'usine à liaisons gérant les liaisons entre le composant client et le serveur.
4. La méthode est appelée directement dès qu'on récupère l'interface correspondante.

3.4.1.4 Outils d'aide à la construction de systèmes

L'architecture THINK met a disposition du programmeur système des outils pour faciliter le développement. Le programmeur peut notamment décrire l'ensemble des dépendances entre les composants dans des fichiers XML faciles à lire et maintenir. Un outil fourni par l'architecture THINK est alors capable de générer le graphe de composition du système à partir de ces fichiers de description de l'architecture. Ce graphe permet notamment de trouver l'ordre dans lequel doivent être initialisé les composants au démarrage du système. L'édition de liens est également automatisée de façon à n'inclure dans le fichier binaire correspondant au système que les composants nécessaires. Des vérifications sont également effectuées comme la détection des cycles dans le graphe de composition ou pour s'assurer que toutes les interfaces requises sont bien implantées. La figure 3.8 illustre un exemple d'un fichier de configuration XML pour un composant ordonnanceur de processus implantant la politique d'ordonnancement connue sous le nom de *Round robin*.

La première ligne précise le nom du composant et sa classe, qui sert à regrouper les composants par thème. La balise `file` permet de préciser le nom du fichier `.c` contenant le programme du composant. La balise `produces` précise le nom de la ou des interfaces implantées par le composant. Enfin, la balise `consumes` permet de lister les interfaces (c'est à dire les services) dont dépend le composant. Dans le cas de cet ordonnanceur, on voit que l'interface qu'il exporte s'appelle `scheduler` et qu'il a besoin de composants implantant

```
<component name="roundrobin" class="cpu">
  <file name="services/cpu/roundrobin.powerpc"/>
  <produces interface="scheduler"/>
  <consumes interface="allocator"/>
  <consumes interface="space"/>
  <consumes interface="trader"/>
  <consumes interface="BF"/>
  <consumes interface="FW"/>
  <consumes interface="trap"/>
</component>
```

FIG. 3.8 – Exemple de fichier de description d'un composant ordonnanceur

des services d'allocation de mémoire, d'espace d'adressage, de courtage, d'usine à liaisons, d'accès au BIOS de la machine et de gestion des interruptions pour fonctionner.

Un composant THINK est implanté selon un modèle classique peu contraignant. Chaque composant doit implanter une méthode d'initialisation, classiquement appelée *composant*`Probe` où *composant* est le nom du composant implanté. Ces méthodes seront appelées au démarrage du système dans l'ordre préconisé par le graphe de composition généré à la compilation du système.

Les interfaces dans THINK sont écrites en Java [GJSB00]. Un outil fourni par l'architecture permet de générer à la compilation les fichiers d'en-tête `.h` nécessaires à partir de ces interfaces Java. La figure 3.9 illustre l'exemple de l'interface de l'ordonnanceur de processus vu plus haut.

```
package services;
public interface Scheduler {
  public Job newJob(Space space,
                    int prio);
  public void destroyJob();
  public Job getJob();
  public Space getSpace();
  public void yield();
  public void wait(int[] lock);
  public int waitto(int[] lock,
                    int timeout);
  public void notify(int[] lock);
}
```

FIG. 3.9 – Exemple d'interface Java d'un composant ordonnanceur

3.4.2 Évaluation

3.4.2.1 Flexibilité

Le principal point fort de l'architecture THINK est sa flexibilité. En effet, en n'imposant aucune abstraction au programmeur on lui permet de construire son système exactement comme il le désire. Parallèlement, le fait de proposer une bibliothèque de composants systèmes modulaires et indépendants permet d'éviter au programmeur d'avoir à réécrire tous les services dont il aura besoin comme c'est le cas avec un nano-noyau simple. Cette bibliothèque ne compromet en rien la flexibilité de l'architecture puisque le programmeur système peut remplacer n'importe quel composant système fourni par un autre programmé par ses soins s'il le désire.

L'architecture THINK est particulièrement intéressante pour la construction de systèmes adaptables. Le mécanisme de liaisons dynamiques supporte le remplacement d'un composant par un autre ce qui permet d'adapter le système dynamiquement en changeant un composant proposant un service donné par un autre plus adapté au nouvel environnement par exemple. De plus, le courtier permet de rendre les changement transparents pour les applications, puisque celles-ci désignent les composants qu'elles utilisent par leur nom symboliques et ne sont pas dépendantes de l'implantation des composants tant qu'ils proposent une interface équivalente. Bien évidemment, le remplacement d'un composant ne peut se faire n'importe quand, notamment pas pendant qu'une application est en train d'exécuter une méthode du composant. De plus, on doit pouvoir s'assurer que le nouveau composant est bien compatible avec les applications qui vont l'utiliser. L'architecture THINK fournie pour cela un ensemble d'outils d'aide à la reconfiguration dynamique du noyau construit. Ces mécanismes sont présentés de façon détaillée dans [Sen03].

3.4.2.2 Performances

La flexibilité de l'architecture THINK entraîne une augmentation des performances par rapport à des systèmes moins flexibles. Sur le plan de la vitesse d'exécution tout d'abord puisque le programmeur peut optimiser son système pour exploiter au mieux le matériel sous-jacent, par exemple en remplaçant un pilote de périphérique générique par un pilote optimisé pour son modèle de matériel (les cartes graphiques sont un exemple typique de matériel pour lequel ce type d'optimisations peut augmenter de façon très importante les performances). De plus, il a été prouvé dans [Fas01] que le passage par le canevas logiciel de THINK pour tous les appels de méthodes inter-composants n'entraînaient pas de dégradation significative des performances. Enfin, il est bien évident que le fait de n'inclure que les abstractions dont on a besoin évite d'encombrer la mémoire, ce qui laisse donc plus de mémoire vive aux applications et au système pour fonctionner dans des conditions optimales.

3.4.2.3 Sécurité

L'implantation de l'architecture THINK réalisée sur PowerPC n'intègre à l'heure actuelle aucun mécanisme de sécurité. Par exemple, les appels inter-segments ne sont pas gérés ce qui empêche d'isoler les applications dans des segments différents. De même, le canevas logiciel de THINK n'est absolument pas sécurisé puisque rien n'empêche une application malveillante de le court-circuiter en fabricant un pointeur pour appeler directement une méthode. Une partie du travail présenté dans ce rapport consiste donc à sécuriser le canevas logiciel de THINK en montrant qu'on peut le faire tout en préservant sa flexibilité initiale.

3.5 Conclusion

De nombreux travaux ont été conduits pour améliorer la flexibilité des systèmes d'exploitation depuis les tout premiers systèmes multi-utilisateurs. Cependant, cette amélioration de la flexibilité se traduit généralement par une dégradation d'une autre propriété du système, notamment la sécurité dans le cas des noyaux extensibles, les performances pour les micro-noyaux, ou la complexité de mise en service du système pour les nano-noyaux où le programmeur doit implanter lui-même toutes les abstractions dont il a besoin.

La recherche concernant l'amélioration de la flexibilité des systèmes est également très active, comme le montrent les nombreux prototypes que nous avons présentés. De façon assez similaire aux architectures classiques, cette recherche d'une plus grande flexibilité du noyau se fait le plus souvent au détriment des performances ou de la sécurité globale du système. Certaines pistes nous paraissent néanmoins intéressantes et méritent d'être approfondies. Ainsi, l'idée de séparer la gestion de la politique de sécurité des outils servant à sa mise en œuvre telle qu'elle est soutenue dans les systèmes Hydra et DTOS nous semble une bonne méthode pour garantir la flexibilité de la protection d'un système. La philosophie de l'exo-noyau consistant à n'imposer aucune abstraction au programmeur tout en lui fournissant une bibliothèque de services utilisables à sa guise permet de bénéficier de la flexibilité de l'architecture nano-noyau tout en bénéficiant d'un minimum de support lors de la construction du système qui évite d'avoir à implanter tous les services soi-même. L'exo-noyau met aussi en évidence, de par ses propres limitations, la nécessité de fournir au constructeur du système un modèle de programmation homogène pour la programmation de ses propres services et applications. L'architecture OSKit nous montre l'intérêt d'une bibliothèque de composants modulaires pouvant être assemblés selon les besoins du programmeur et souligne la nécessité de pousser jusqu'au bout cette volonté de modularisation pour garantir une véritable indépendance des modules et donc la meilleure flexibilité possible pour le système. Les prototypes Scout et Escort définissent la notion de chemin d'entrée/sortie qui permet d'assurer la facturation chaînée des consommations

de ressources du système. Cette notion peut être particulièrement intéressante pour la protection du système contre les attaques contre la qualité de service du système puisqu'elle permet de retrouver tous les intervenants dans une consommation de ressources et donc potentiellement le consommateur initialement fautif lors d'une attaque.

Ces différentes pistes nous amènent à l'étude de l'architecture de systèmes flexibles THINK. En effet, elle répond à la plupart des besoins exprimés lors de l'étude des différents prototypes de recherche présentés. Elle respecte la philosophie de l'exo-noyau en étant construite autour d'un noyau minimal n'imposant aucune abstraction au programmeur tout en lui fournissant une bibliothèque de services systèmes. Elle propose un modèle de programmation par composants pouvant être utilisé pour développer aussi bien des services systèmes que des applications. Elle garantit l'indépendance des composants du point de vue de leur implantation en ne permettant la dépendance d'un composant que vis à vis de l'interface d'un autre. Toutes ces caractéristiques en font selon nous une plate-forme idéale pour la mise en œuvre de mécanismes de protection flexibles inspirés de ceux trouvés dans les systèmes classiques. Notre démarche consiste donc à sécuriser l'architecture THINK en montrant qu'on ne compromet pas ce faisant la flexibilité originale du système. Pour cela, on commence par intégrer à la bibliothèque de THINK des outils élémentaires de protection que nous détaillons au chapitre suivant.

Chapitre 4

Outils élémentaires de sécurité

4.1 Introduction

Un des objectifs de notre travail est de montrer qu'il est possible de concilier flexibilité et sécurité en fournissant des mécanismes de sécurité ne compromettant pas la flexibilité du système. Pour cela, nous présentons dans ce chapitre des outils élémentaires, répondant chacun à un problème de sécurité précis, et montrons qu'ils peuvent être implantés sans imposer de contraintes sur la politique de sécurité que le programmeur choisira de mettre en œuvre dans son système. Nous présenterons dans le chapitre suivant le canevas logiciel permettant d'utiliser ces outils de façon sécurisée et flexible. Le gestionnaire de sécurité est un élément de ce canevas logiciel auquel nous faisons référence à plusieurs reprises dans ce chapitre. Ce composant sert à mettre en œuvre la politique de sécurité choisie par le programmeur système, comme cela sera explicité au chapitre suivant.

Nous avons choisi d'étudier trois ressources fondamentales d'un ordinateur, à savoir la mémoire, le disque dur et le réseau. Pour chacune de ces ressources, nous avons prototypé des outils élémentaires permettant de faire face à un type d'attaque donné. Il est important de noter que ces outils n'ont pas été développés dans le but d'offrir des protections parfaites contre ces attaques. Les problématiques de l'isolation mémoire, d'ordonnancement du disque dur et de protection contre les attaques par déni de service réseau sont des sujets très complexes pour lesquels beaucoup de travaux ont déjà été effectués et beaucoup restent certainement à faire. Le but de ce travail est de montrer qu'il est possible d'implanter des outils de sécurité réalistes et fonctionnels tout en préservant la flexibilité de gestion de la politique de sécurité. On s'est donc attaché, tout en montrant que ces outils assurent une fonction de sécurisation élémentaire, à préciser systématiquement la séparation entre la partie fonctionnelle réalisée par l'outil, et la partie gestion de la politique mise en œuvre par le gestionnaire de sécurité.

4.2 Isolation mémoire

4.2.1 Problématique

L'isolation mémoire est un problème difficile et pour lequel de nombreuses solutions très diverses ont été proposées. L'isolation matérielle pose un problème de flexibilité et nécessite évidemment de disposer d'un matériel offrant les caractéristiques voulues. L'interprétation quand à elle pose un problème d'efficacité et se révèle peut adaptée pour les noyaux de systèmes d'exploitation. Nous avons donc fait le choix d'implanter un outil d'isolation mémoire logicielle afin de conserver un maximum de flexibilité dans la gestion de la politique d'isolation.

4.2.2 Principe de l'outil d'isolation mémoire

Cet outil est basé sur la technique d'injection de code présentée dans [WLAG93] sous le nom de correspondance de segments (de l'anglais *segment matching*). Son principe est de parcourir le code de chaque processus au moment de sa création pour remplacer tous les accès mémoire par un branchement vers un point d'entrée « bien-connu » dans le gestionnaire de sécurité. Ce point d'entrée correspond en fait à une méthode dont le rôle est de vérifier que l'accès mémoire en question est bien autorisé par la politique de sécurité. Si c'est le cas l'instruction remplacée est exécutée et le pointeur d'exécution est ramené sur l'instruction suivant celle remplacée. Dans le cas contraire une exception de sécurité est levée. Le fonctionnement de cet outil est illustré dans la figure 4.1.

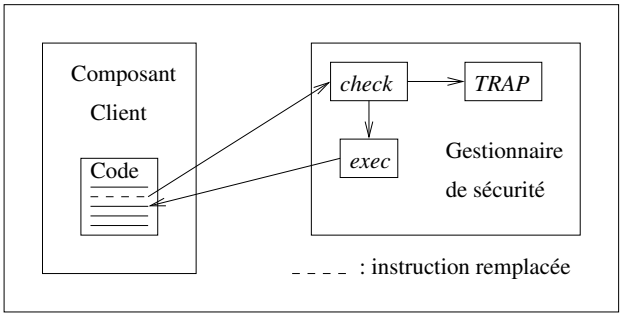

FIG. 4.1 – Principe de fonctionnement de l'outil d'isolation logicielle

Par rapport à l'algorithme présenté dans [WLAG93], cet outil se révèle beaucoup plus flexible car la vérification effectuée par le gestionnaire de sécurité est complètement libre et dépend de la politique de sécurité, alors que l'algorithme original de correspondance de segment n'est capable que de vérifier si l'accès a lieu dans une zone autorisée. A noter également que notre outil permet de choisir quelles instructions doivent être vérifiées, ce qui permet par exemple de ne vérifier que les branchements et pas les lectures ou écritures

en mémoire si c'est ce qu'exige la politique de sécurité choisie.

4.2.3 Instantiation pour la correspondance de segments

Comme on l'a dit, l'outil implanté reste complètement indépendant de la politique de sécurité à mettre en œuvre, puisque la vérification de la légitimité de l'accès mémoire est effectuée dans le gestionnaire de sécurité. Pour tester cet outil, nous avons défini une politique très simple implantant la correspondance de segments présentée dans [WLAG93]. Nous avons choisi d'implanter cette politique particulière car sa mise en œuvre se révèle très simple et concise, et permet d'espérer de bonnes performances qui pourront servir de comparaison pour des politiques plus complexes.

4.2.3.1 Fonctionnement

On souhaite donc vérifier que l'adresse de destination des accès mémoire du processus sont compris dans un intervalle défini par les bornes inférieures et supérieures de la zone. On prend ici l'exemple simple d'une instruction de chargement d'un entier sur 32 bits de la mémoire vers un registre, comme illustré par la figure 4.2.

```
Initial_code:
  lwz 1,8(2)
```

FIG. 4.2 – Exemple d'une instruction de chargement d'un entier

Ce code charge dans le registre 1 l'entier stocké à l'adresse calculée en ajoutant 8 au contenu du registre 2.

Ce code est transformé par l'outil d'isolation en un branchement vers le point d'entrée dans le gestionnaire de sécurité comme illustré par la figure 4.3.

```
Modified_code:
  ba Entry_Point_LWZ
```

FIG. 4.3 – Code modifié par l'outil d'isolation mémoire

Puisque nous implantons une simple correspondance de segments, le code de vérification dans le gestionnaire de sécurité peut être généré dynamiquement comme pour l'algorithme original présenté dans [WLAG93]. Lorsque l'outil parcourt le code du processus pour remplacer les accès mémoire, il génère le code de vérification dans une zone mémoire appartenant au gestionnaire de sécurité. Ce code est fonction de l'instruction remplacée, sachant que toutes les instructions du même type nécessitent la génération du même code de vérification (mise à part pour l'instruction elle-même dont les arguments

peuvent différer d'une occurrence à l'autre). La figure 4.4 détaille le code généré pour l'exemple de l'instruction de chargement d'un entier.

```
Verification_code:
  add 14,2,8
  tw 8,14,15
  tw 16,14,16
  lwz 1,8(2)
  ba Modified_Code + 4
```

FIG. 4.4 – Code généré par l'outil d'isolation mémoire

L'instruction `add` calcule l'adresse de destination de l'accès mémoire et la stocke dans le registre 14. Les deux instructions `tw` comparent cette adresse aux bornes de la zone autorisée (qui sont stockées dans les registres 15 et 16) et lèvent une exception si l'adresse n'est pas dans les bornes. L'instruction `lwz` est l'instruction de chargement remplacée qui est donc exécutée une fois que l'adresse a été vérifiée. Enfin, l'instruction `ba` permet de revenir à l'instruction suivant celle qui a été remplacée.

Optimisation Une optimisation simple de cet algorithme consiste à ne pas générer de code pour les accès mémoire dont il est possible de calculer l'adresse statiquement (c'est à dire au moment où l'on crée le processus). Cela est possible par exemple pour certains branchements relatifs à la position actuelle du pointeur de programme, comme dans l'instruction `b 2` qui incrémente le pointeur de programme de 8 octets. Pour ces instructions, il est donc possible de vérifier immédiatement si l'adresse de destination est autorisée et dans le cas contraire de lever une exception. Cette optimisation est en pratique très utile car les branchements relatifs sont très fréquents dans la plupart des programmes (notamment pour les boucles).

Il est à noter que pour mettre en œuvre une politique plus complexe que celle présentée dans cet exemple simple, on devra certainement implanter dans le gestionnaire de sécurité un code de vérification plus long et plus coûteux que celui nécessaire à la correspondance de segment. On aura cependant tout intérêt à optimiser ce code car il sera exécuté pour chaque accès mémoire à vérifier.

4.2.3.2 Performances

Nous avons mesuré les performances de notre outil pour l'exemple de la correspondance de segments.

Consommation mémoire Générer du code de vérification consomme de la place en mémoire. L'espace consommé dépendra bien évidemment du nombre d'instructions à

vérifier et donc de la fréquence des accès mémoire dans le processus. Comme la plupart des compilateurs modernes sont capables d'optimiser le code pour réduire autant que possible le nombre d'accès mémoire (notamment en utilisant les registres du processeur pour stocker les variables locales afin d'éviter les accès dans la pile), la taille du code généré reste en général raisonnable. Par exemple, pour un programme de tri à bulles (donc un programme faisant beaucoup d'accès mémoire), on compte 9 instructions effectuant des accès mémoire sur 29 instructions au total. Sur ces 9 instructions, 4 sont des branchements relatifs pouvant être vérifiés statiquement. On génère donc du code de vérification pour 5 instructions, soit 25 instructions de vérification. On double donc la taille du code pour un programme effectuant beaucoup d'accès mémoire, c'est à dire dans le pire des cas. Si on se rappelle que dans la plupart des programmes la taille du code est très inférieure à celle des données, cette consommation mémoire paraît raisonnable.

Temps de génération Pour mesurer le temps de génération du code de vérification à la création du processus à isoler, nous avons utilisé un programme test de 29000 instructions dont 9000 accès mémoire. Le temps de génération obtenu est de 3.20 ms alors que le temps de lecture de l'exécutable sur le disque et de création du processus est de l'ordre de 16.5 ms. L'algorithme engendre donc un surcoût d'environ 20% à la création du processus, ce qui semble acceptable pour la plupart des programmes.

Pénalité à l'exécution Exécuter un code de vérification à chaque exécution d'un ou plusieurs types d'instruction a forcément un impact sur les performances du processus. Nous avons donc effectué des tests sur différents types de programmes pour déterminer les cas pour lesquels la correspondance de segments s'avérait intéressante et ceux pour lesquels elle est trop coûteuse pour être employée de façon systématique.

Instruction de chargement/déchargement mémoire On observe une multiplication du temps d'exécution par 3.5 pour l'exécution d'une instruction de lecture ou d'écriture en mémoire. Ce résultat peut paraître prohibitif mais il faut garder à l'esprit qu'un programme n'est bien évidemment jamais composé que d'accès mémoire.

Instruction de branchement relatif Grâce à l'optimisation présentée précédemment, il n'y a aucun surcoût à l'exécution pour toutes les accès mémoire dont l'adresse de destination peut être calculée statiquement.

Instruction de branchement absolu Ce type d'instruction est notamment utilisé pour les appels inter-processus, qui sont en général très coûteux lorsqu'on utilise un mécanisme d'isolation matérielle puisque l'appel direct est impossible et qu'on doit passer par un mécanisme comme les *sockets* par exemple. Nous avons donc comparé trois types

d'appels inter-processus : un appel direct sans isolation mémoire, un appel direct via une isolation logicielle, et un appel en utilisant les *sockets Unix* pour contourner une isolation matérielle. La fonction appelée est vide ce qui permet de mesurer le temps d'appel. Les résultats de ces tests sont présentés dans la Figure 4.5.

	Aucune isolation	Isolation logicielle	Isolation matérielle
Temps d'appel	0.025 μs	0.045 μs	7.312 μs
Surcoût	nul	$\times$1.80	$\times$292.48

FIG. 4.5 – Performances d'un branchement absolu

Le surcoût est calculé par rapport à un appel direct dans un système sans isolation. On voit que l'isolation logicielle entraîne un doublement du temps d'appel à la fonction alors que l'utilisation des *sockets Unix* rend cet appel près de 300 fois plus lent que l'appel direct sans isolation. L'isolation logicielle se révèle donc particulièrement adaptée pour les appels inter-processus.

Tri à bulles Ce programme est un exemple typique de logiciel effectuant beaucoup d'accès mémoire à l'exécution. Afin de maximiser ce nombre d'accès mémoire, nous avons volontairement désactivé les optimisations du compilateur qui permettent de déplacer les variables locales du programme dans des registres pour éviter les accès à la pile. Pour le tri de 100000 entiers, on obtient un surcoût à l'exécution de +98% soit un tri deux fois plus lent que sans isolation.

Calcul de racine carrée Ce programme est a contrario un exemple de programme faisant très peu d'accès mémoire (l'algorithme utilisé est celui d'Héron d'Alexandrie, basé sur la méthode dite des moyennes arithmétiques). Le surcoût pour le calcul de la racine carrée de 100000 est de +1.55%, ce qui est négligeable.

GZip Ce programme est fréquemment utilisé sous UNIX pour compresser des données. C'est un programme faisant beaucoup d'accès mémoire, notamment des manipulations de chaînes de caractères dans un dictionnaire. Le surcoût pour la compression de 128 Mo de données est de +117%, d'où une compression plus de deux fois plus lente.

Conclusion L'analyse de ces performances indique clairement que, de façon très logique, le surcoût à l'exécution du mécanisme d'isolation est très dépendant de la fréquence des accès mémoire dans le processus isolé. Un surcoût de +1.55% est parfaitement négligeable, par contre un doublement du temps d'exécution d'un processus aura un impact sensible sur les performances du système. Il appartient donc au programmeur système de choisir avec soin les applications pour lesquelles il désire appliquer cet outil d'isolation et quel type

d'instructions il juge nécessaire de vérifier. On notera les performances très intéressantes concernant les appels inter-processus, particulièrement comparées à celles d'un appel inter-processus via une isolation matérielle, qui mettent en évidence l'adéquation de ce type d'isolation pour les systèmes composés de nombreux modules fournissant des services inter-dépendants. De façon plus générale, l'indépendance complète de l'outil d'isolation par rapport à la politique de sécurité choisie et son caractère complètement optionnel sont un atout pour le programmeur qui peut donc effectuer ses choix de politiques d'isolation en fonction des applications visées.

4.3 Gestionnaire de disque adaptable

4.3.1 Problématique

L'ordonnancement des requêtes de lecture et d'écriture sur disque dur est un problème pour lequel de nombreuses solutions ont été proposées. Aucun algorithme ne s'est cependant imposé, principalement à cause des différentes contraintes que l'on peut trouver dans un système. En effet, un algorithme efficace pour un système destiné à des applications multimédia ne sera pas forcément adapté à un système embarqué par exemple. L'algorithme d'ordonnancement, c'est à dire la façon de trier les requêtes dans la file dans laquelle elles sont stockées en attendant d'être traitées en rafale, est en fait l'expression d'une politique de gestion d'une ressource, la bande-passante disque comme on la désigne parfois par analogie avec la bande-passante réseau. Il est donc essentiel de dissocier cette politique des outils qui servent à la mettre en œuvre si l'on souhaite conserver la flexibilité du système. Ce n'est généralement pas le cas dans la plupart des systèmes actuels, pour lesquels la politique d'ordonnancement est programmée dans l'ordonnanceur lui-même et ne peut être remplacée. Nous avons donc développé un gestionnaire de disque adaptable, qui permet de changer la politique d'ordonnancement dynamiquement en fonction des besoins du système.

4.3.2 Implantation

La version de l'architecture THINK sur laquelle nous avons travaillé ne fournissait pas d'ordonnanceur de disque (toutes les requêtes étant traitées immédiatement par le gestionnaire de disque dans leur ordre d'arrivée). Nous avons donc implanté un composant ordonnanceur de disque gérant une file des requêtes, comme détaillé dans la figure 4.6.

La structure `DiskRequest` décrit une requête d'accès à un secteur du disque. On notera la présence de l'information permettant d'identifier le processus émetteur de la requête dans cette structure. Cette information n'est pas disponible dans la plupart des gestionnaires classiques, mais on l'inclut ici car elle peut être nécessaire pour certains

```
typedef struct {
  int type;                 // Lecture ou Ecriture
  unsigned long int sector; // Secteur à accéder
  char *buffer;             // Zone mémoire où stocker les données lues
  unsigned int pid;         // Identifiant du processus emetteur de la requête
} DiskRequest;

DiskRequest *queue;

extern void initQueue();
extern char *readSector(unsigned long int sector,
                        char *buffer, unsigned int pid);
extern void writeSector(unsigned long int firstSector,
                        char *buffer, unsigned int pid);
extern void flushQueue();
extern DiskRequest *getQueue();
extern void setScheduling(void (*sort)());
extern void removeRequest(DiskRequest *request);
```

FIG. 4.6 – Interface d'un ordonnanceur de disque

types d'ordonnancement. La méthode `initQueue` permet d'initialiser la file des requêtes et les méthodes `readSector` et `writeSector` permettent aux applications d'émettre des requêtes de lecture et d'écriture vers les secteurs voulus du disque. La méthode `flushQueue` sert à exécuter les requêtes stockées dans la file, c'est donc la méthode qui sera appelée périodiquement par le processus en charge de la gestion des accès disques afin d'exécuter les requêtes en rafale.

La méthode `getQueue` est notre premier outil élémentaire de gestion du disque, qui permet au gestionnaire de sécurité d'avoir une vue des requêtes dans la file et donc de pouvoir baser ses décisions de gestion de la politique d'ordonnancement en fonction des requêtes actuellement dans la file.

La méthode `setScheduling` est le deuxième outil élémentaire implanté. Il permet au gestionnaire de sécurité de communiquer à l'ordonnanceur de disque l'algorithme d'ordonnancement à appliquer, qui est passé en paramètre sous forme d'un pointeur de fonction. La fonction en question a pour effet de réorganiser les requêtes dans la file en fonction de la politique choisie. Il est important que la réorganisation des requêtes dans la file soit la plus courte possible puisqu'une réorganisation des requêtes prenant trop de temps pourrait à l'extrême annuler le bénéfice de l'ordonnancement.

La méthode `removeRequest` est le troisième outil implanté et permet au gestionnaire de sécurité de supprimer une requête de la file s'il le juge nécessaire. Cet outil peut notamment être utilisé pour contrer une attaque par déni de service lancée par un processus qui émettrait beaucoup de requêtes pour consommer toute la bande-passante disque. Ces trois outils restent complètement indépendants de la politique de gestion de la ressource disque

choisie puisque que c'est le gestionnaire de sécurité qui fixe l'algorithme d'ordonnancement et qui décide éventuellement de supprimer des requêtes de la file. Les outils implantés n'ont donc pas d'autre fonction que de permettre au gestionnaire de sécurité d'appliquer la politique choisie.

4.4 Gestionnaire de réseau sécurisé

4.4.1 Problématique

Comme on l'a vu précédemment, les attaques par inondation de requêtes sont un problème typique des serveurs offrant des services sur Internet. Pour illustrer notre travail sur la sécurisation de la ressource réseau, nous avons donc implanté un certain nombre d'outils élémentaires permettant d'améliorer la résistance du système à ces attaques.

4.4.2 Filtrage des paquets

Le premier outil implanté est un simple filtre qui accepte ou non les paquets (SYN ou autres) en fonction de l'adresse IP de l'émetteur. La comparaison est effectuée entre l'adresse encodée dans le paquet et une liste d'adresse bannies gérée par le gestionnaire de sécurité. Cette liste peut évoluer dynamiquement en fonction des attaques pour ajouter de nouvelles adresses. Il faut noter que cette protection n'est pas parfaite car l'adresse IP encodée dans le paquet SYN reçu peut être falsifiée, voire même changer pour chaque paquet, mais elle permet de résister aux flux de paquets SYN provenant de la même adresse. En effet, le gestionnaire de sécurité, détectant une attaque, peut ajouter l'adresse émettrice dans la liste des adresses bannies, ce qui aura pour effet de neutraliser cette attaque même si cette adresse émettrice a été falsifiée et ne correspond en réalité pas à la machine émettant l'attaque. On notera les deux partie distinctes du mécanisme : l'outil, indépendant de la politique, qui est en fait une simple fonction permettant au gestionnaire de sécurité d'accéder à l'adresse IP source stockée dans le paquet TCP reçu, et le gestionnaire de sécurité, qui définit la politique de sécurité c'est à dire ici la liste des adresses IP bannies et renvoie au gestionnaire TCP l'autorisation ou non d'accepter le paquet.

4.4.2.1 Implantation

L'implantation de cet outil se révèle très simple puisqu'il suffit de rajouter quelques lignes dans le gestionnaire TCP comme l'illustre la figure 4.7.

Comme on peut le voir, le gestionnaire de sécurité, appelé par `CALL1(SecurityManager, isBanned, saddr))`, renvoie un booléen pour indiquer au gestionnaire TCP que l'adresse émettrice est bannie.

```
// gestionnaire dérivé de Linux 2.4.19 (net/ipv4/tcp_ipv4.c)
int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb) {
...
  banned = *((int *) CALL1(SecurityManager, isBanned, saddr));
  if (banned) goto drop;
...
```

FIG. 4.7 – Implantation du filtrage des paquets dans le gestionnaire TCP

4.4.2.2 Performances

Nous avons soumis cet outil à une inondation de paquets grâce à un outil de génération de paquets que nous avons implanté[1]. Le gestionnaire TCP que nous utilisons est dérivé du gestionnaire de Linux 2.4.19, et a été écrit de façon à éviter les dépassements de capacité de la file de stockage en cas d'inondation (ce qui signifie que si la file est pleine, tous les nouveaux paquets arrivants sont rejetés). Soumis à une inondation de 50000 paquets par secondes, le système est considérablement ralenti et aucune demande de connexion légitime n'est plus acceptée. Avec notre outil, aucune perte de performance n'est visible (la différence est flagrante pour un utilisateur écrivant à la console du système : dans le premier cas les caractères mettent plusieurs secondes à s'afficher alors qu'ils le font instantanément dans le deuxième) et les connexions légitimes s'établissent normalement.

4.4.3 Surveillance de la file de stockage

Pour éviter un débordement de capacité de la file de stockage, nous avons implanté un outil permettant de connaître le taux d'occupation de la file et de la vider en cas de saturation. Le gestionnaire de sécurité peut donc vérifier que le nombre de paquets dans la file ne dépasse pas un seuil défini par la politique de sécurité et prendre la décision de vider la file si sa taille augmente de façon dangereuse. L'outil est encore une fois indépendant de la politique de sécurité puisqu'il s'agit de deux simples fonctions renvoyant le nombre de paquets dans la file pour la première et vidant la file pour la deuxième. On notera que vider la file des requêtes aura certainement pour effet de rejeter des requêtes légitimes, mais c'est évidemment moins grave que de risquer un dépassement de capacité de la file ou une saturation de la mémoire du serveur.

4.4.3.1 Implantation

Là encore, l'implantation de cet outil est très simple comme illustré par la figure 4.8.

[1]Nous ne donnerons pas ici le code de cet outil pour éviter qu'il soit utilisé par des personnes mal-intentionnées pour lancer des attaques sur Internet. Il est dérivé de l'utilitaire d'analyse de pare-feux bien connu *nmap*. Pour information, la modification du code de *nmap* pour le transformer en générateur de paquets nous a pris 15 minutes et nécessite la modification de 5 lignes de code C. Cela illustre le réel danger posé par les attaques par déni de service vu la simplicité de leur mise en œuvre.

```
// gestionnaire dérivé de Linux 2.4.19 (net/ipv4/tcp_ipv4.c)
int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb) {
...
  nbrSYN = sk->tp_pinfo.af_tcp.listen_opt->qlen_young;
  emptyQueue = *((int *) CALL1(SecurityManager, checkNbrSYN, nbrSYN));
  if (emptyQueue) {
    for (i = sk->tp_pinfo.af_tcp.listen_opt->qlen_young; i > 0; i --) {
      tcp_synq_removed(sk,
                       sk->tp_pinfo.af_tcp.listen_opt->syn_table[i - 1]);
    }
  }
...
```

FIG. 4.8 – Implantation de la surveillance de la file de stockage dans le gestionnaire TCP

L'appel au gestionnaire de sécurité est réalisé par `CALL1(SecurityManager, checkNbrSYN, nbrSYN)`. Cet appel renvoie un booléen, `emptyQueue`, indiquant si la file doit être vidée ou non. Le vidage de la file est réalisé grâce à la fonction `tcp_synq_removed` existant dans le gestionnaire TCP initial et permettant de supprimer un élément donné de la file de stockage.

4.4.3.2 Performances

Le gestionnaire TCP extrait de Linux 2.4.19 est déjà pourvu d'un mécanisme permettant d'éviter un dépassement de capacité de la file de stockage en rejetant tout nouveau paquet SYN dès que la file est pleine. Nous avons donc réglé notre outil de façon a vider la file lorsqu'elle atteint les 3/4 de sa capacité, de façon à conserver de l'espace dans la file pour des requêtes de connexions privilégiées (par exemple provenant d'une plage d'adresses donnée ou étiquetées avec l'option TCP `urgent`). En modifiant légèrement le code présenté dans la figure 4.8, on peut ne supprimer que les requêtes non privilégiées et ainsi garantir à celles-ci de bonnes chances de connexion même pendant une attaque. En pratique, lorsqu'on soumet le gestionnaire à une attaque par inondation comme présentée pour l'outil précédent, on vérifie qu'une requête de connexion venant d'une adresse IP privilégiée aboutit même pendant une attaque alors que celles venant d'adresses non privilégiées ne réussissent que de façon aléatoire (c'est à dire si elles arrivent « au bon moment », avant que le taux d'occupation de la file ne dépasse le seuil autorisé).

4.4.4 Délocalisation de la file de stockage

Le danger de dépassement de capacité de la file de stockage provient du fait que cette file est allouée dans l'espace d'adressage du gestionnaire réseau qui fait partie du noyau du système d'exploitation, pour lequel il n'y a en général pas de quota limitant la quantité de mémoire utilisable. Si la file de stockage était allouée dans l'espace applicatif, par

exemple l'espace d'adressage du serveur HTTP, alors un dépassement de capacité aurait des conséquences beaucoup moins graves car il ne provoquerait au pire qu'un arrêt du serveur et pas de tout le système. Il est de plus possible au noyau de contrôler la quantité de mémoire consommée par une application et d'empêcher que celle-ci ne sature la mémoire de la machine. Ce mécanisme nécessite une modification de la bibliothèque de gestion des *sockets* pour permettre à un processus de donner l'adresse de sa file de stockage propre lors de la création d'une *socket* serveur. L'outil élémentaire est donc un module de gestion des *sockets* TCP qui permet l'allocation de la file de stockage dans l'espace d'adressage des applications qui l'utilisent. Cet outil reste indépendant de la politique de sécurité car il permet simplement de choisir l'endroit où sera allouée la file de stockage.

4.4.4.1 Implantation

Implanter cet outil nécessite une simple modification du gestionnaire TCP comme illustré par la figure 4.9.

```
// gestionnaire dérivé de Linux 2.4.19 (net/ipv4/tcp.c)
int tcp_listen_start(struct sock *sk) {
...
// On remplace la ligne :
//lopt = kmalloc(sizeof(struct tcp_listen_opt), GFP_KERNEL);
// qui alloue la file dans l'espace d'adressage du noyau par :
  lopt = (char *) CALL1(UserMemoryManager, alloc,
                        sizeof(struct tcp_listen_opt));
// qui l'alloue dans l'espace utilisateur
// (géré ici par le composant UserMemoryManager)
...
```

FIG. 4.9 – Implantation de la délocalisation de la file de stockage dans le gestionnaire TCP

Le composant `UserMemoryManager` est utilisé dans cet exemple pour symboliser le gestionnaire de l'espace mémoire utilisateur. Dans un système Unix classique, cela se traduira tout simplement par l'utilisation de la fonction `malloc` en lieu et place de la fonction noyau `kmalloc`.

4.4.4.2 Performances

Cet outil n'entraîne pas de modification des performances si le gestionnaire TCP peut accéder directement à l'espace utilisateur. En effet, si un accès mémoire effectué par le gestionnaire TCP est plus coûteux vers l'espace utilisateur que vers l'espace noyau, par exemple à cause d'un mécanisme d'isolation mémoire, alors ce mécanisme aura forcément des conséquences sur les performances du gestionnaire. Dans le cas contraire il est transparent.

4.4.5 Contrôle du débit entrant

Pour éviter la saturation de la file de stockage, on peut essayer de contrôler le nombre de paquets SYN arrivant sur le serveur. Pour cela, on fournit un outil capable d'établir des statistiques en temps réel sur le nombre de paquets reçus par seconde. Le gestionnaire de sécurité peut donc connaître le débit entrant et en fonction de la politique de sécurité, décider si ce débit correspond à une attaque et réagir en utilisant un outil implanté dans le gestionnaire TCP qui rejettera aléatoirement des paquets SYN afin d'en réduire le débit entrant. Cette technique engendrera elle aussi vraisemblablement des pertes de connexions légitimes, mais encore une fois cela est préférable à un dépassement de capacité de la file de stockage. Les deux outils sont complètement indépendants de la politique puisqu'ils consistent simplement à établir des statistiques et à paramétrer le gestionnaire réseau sur décision du gestionnaire de sécurité.

4.4.5.1 Implantation

Cet outil est implanté comme présenté dans la figure 4.10.

```
// gestionnaire dérivé de Linux 2.4.19 (net/ipv4/tcp_ipv4.c)
int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb) {
...
  syn_counter ++;
  rateTooFast = *((int *) CALL1(SecurityManager, synRate, syn_counter));
  if (rateTooFast) goto drop;
  do_gettimeofday(&time_end);
  if (time_end.tv_sec - time_begin.tv_sec > 0) {
    syn_counter = 0;
    do_gettimeofday(&time_begin);
  }
...
```

FIG. 4.10 – Implantation du contrôle du débit entrant dans le gestionnaire TCP

L'appel au gestionnaire de sécurité, réalisé par `CALL1(SecurityManager, synRate, syn_counter)`, permet de l'informer du nombre de paquets SYN reçus et le cas échéant de lui permettre d'ordonner de rejeter les paquets en surnombre. Le compteur de paquets `syn_counter` est réinitialisé toutes les secondes grâce à la fonction `do_gettimeofday` qui permet d'avoir accès au nombre de secondes écoulé depuis le démarrage du système. Ainsi, on obtient un outil capable de calculer en temps réel le nombre de paquets reçu par seconde.

4.4.5.2 Performances

En pratique cet outil permet d'exercer un contrôle fin sur la quantité de paquets SYN traités. La sécurité implantée dans le gestionnaire TCP qui rejette les paquets SYN lorsque

la file est pleine ne permet en effet qu'un contrôle « a posteriori », c'est à dire lorsque la file est pleine et qu'aucune connexion ne peut plus être acceptée. En faisant évoluer le débit seuil en fonction de la vitesse de traitement des requêtes (c'est à dire la vitesse avec laquelle le client répond au paquet SYN-ACK du serveur en envoyant un paquet ACK ce qui a pour effet d'établir la connexion), on peut s'assurer que le gestionnaire TCP ne sera jamais surchargé. Le calcul du débit optimal par le gestionnaire de sécurité s'avérant en pratique complexe et fonction de nombreux paramètres, on ne cherchera pas ici à trouver de formule générale. Dans le cas d'une attaque par inondation, limiter le débit entrant ne permet pas d'assurer la qualité de service du point de vue du réseau car les requêtes légitimes, « noyées » dans les paquets SYN générés par l'attaque, ont en probabilité peu de chance de ne pas être rejetées. Par contre, cet outil s'avère très intéressant du point de vue de la qualité de service global du système puisque, soumis à une attaque similaire à celle décrite lors du test du filtrage des adresses IP ci-dessus, le système ne subit pas de dégradation notable de ses performances du point de vue d'un utilisateur écrivant à la console.

4.5 Conclusion

Comme nous l'avons vu dans ce chapitre, il est possible d'implanter des outils de sécurisation indépendants de la politique de sécurité choisie. Le mécanisme d'isolation mémoire que nous avons développé peut être paramétré pour effectuer n'importe quelle vérification au moment de l'accès mémoire ce qui n'est pas le cas d'une isolation matérielle par exemple. Le gestionnaire de disque adaptable peut appliquer n'importe quel algorithme d'ordonnancement à la différence de l'ordonnanceur d'un système classique comme Linux par exemple où l'algorithme est codé directement dans l'ordonnanceur. Enfin, le gestionnaire de réseau sécurisé peut être paramétré aisément grâce aux différents seuils définis dans le gestionnaire de sécurité. De façon générale, en délocalisant toutes les prises de décision dans ce gestionnaire, on évite de rendre le composant gérant une ressource dépendant de la politique de gestion à mettre en œuvre, à la différence de la plupart des systèmes actuels où la politique de gestion est codée en dur dans le mécanisme de gestion de la ressource. Cette architecture est générique et peut être mise en œuvre dans n'importe quel système fournissant un canevas logiciel sécurisé intégrant un gestionnaire de sécurité, comme celui que nous présentons au chapitre suivant.

Chapitre 5

Canevas logiciel

5.1 Introduction

Le rôle d'un canevas logiciel est de réglementer les interactions entre les différents modules du système. Dans l'architecture THINK, ce canevas logiciel est basé sur un modèle à composants et régit les exportations d'interfaces, les créations de liaisons et les appels de méthodes. Dans le prototype de l'architecture THINK implanté sur PowerPC, ce canevas n'est cependant pas du tout sécurisé.

Cette sécurisation est pourtant nécessaire pour garantir la sécurité globale du système. Les outils élémentaires que nous avons présentés permettent en effet de protéger le système contre des attaques choisies, mais ils n'ont aucune utilité si un composant malveillant peut les court-circuiter ou bien modifier leurs comportements sans contrôle. Par exemple, si le programmeur système utilise notre gestionnaire de disque adaptable pour imposer un algorithme d'ordonnancement garantissant l'équité entre les différents processus, une application malveillante pourrait aisément changer l'algorithme d'ordonnancement pour le remplacer par un algorithme favorisant ses requêtes en utilisant la méthode `setScheduling`.

De plus, la séparation systématique de la gestion de la politique de sécurité des outils servant à sa mise en œuvre requiert la présence dans le système d'un composant se consacrant à la gestion de cette politique. Ce composant, que nous appelons gestionnaire de sécurité, fait partie intégrante du canevas logiciel afin d'assurer une gestion homogène de la protection dans le système. Les communications entre les différents composants applicatifs ou systèmes, et ce gestionnaire de sécurité doivent elles-aussi être sécurisées, de façon à garantir la bonne application des décisions du gestionnaire et donc la mise en œuvre de la politique.

Enfin, concernant la gestion des politiques de sécurité, il est souhaitable de fournir au programmeur système un support pour lui faciliter la tâche lors de la spécification de sa politique. Ce support doit permettre de spécifier simplement des politiques statiques et de

modifier dynamiquement des politiques existantes, tout en garantissant les performances de la gestion des politiques à l'exécution. Ce triple objectif illustre la difficulté du problème dans un noyau de système d'exploitation dont les performances ne peuvent être négligées sous peine de handicaper tout le système.

Nous présentons donc dans ce chapitre la sécurisation du canevas logiciel de l'architecture THINK que nous utilisons pour construire des systèmes flexibles. Nous montrons comment sécuriser cette architecture sans réduire sa flexibilité initiale en implantant un canevas sécurisé respectant le modèle de programmation de l'architecture THINK. Nous détaillons l'implantation de ce canevas et du gestionnaire de sécurité, et donnons des pistes concernant la spécification de politiques de sécurité. Nous décrivons enfin un outil d'aide à la génération de politiques de sécurité statiques adapté à notre implantation de la protection dans le canevas logiciel et montrons qu'il est possible d'appliquer cette technique à la modification dynamique de politiques existantes.

5.2 Sécurisation du canevas logiciel de Think

L'architecture de système flexibles THINK est basée sur un canevas logiciel inspiré du modèle à objet ODP [ODP]. Ce canevas a été défini de façon a maximiser la flexibilité du système et de permettre notamment sa reconfiguration dynamique en permettant le chargement et le déchargement dynamique de composants. A contrario, il n'a pas été défini en pensant aux aspects liés à la sécurité du système. On présente donc dans cette section les objectifs d'un canevas sécurisé, avant de mettre en évidences les principaux manques du canevas actuel en matière de sécurité et de proposer un nouveau canevas sécurisé implantant les mêmes notions que l'ancien.

5.2.1 Objectifs

L'objectif d'un canevas logiciel est de réglementer les interactions entre les composants. Dans le cas d'une architecture basée sur un modèle de programmation comme THINK qui s'inspire du modèle ODP, le canevas logiciel est l'instanciation de ce modèle et doit en intégrer les notions essentielles. Il est toutefois possible que ce canevas instancie des notions qui n'étaient pas présentes dans le modèle sur lequel il est basé. De telles propriétés non-fonctionnelles, comme la protection par exemple, peuvent se révéler indispensable en pratique.

Il est nécessaire de sécuriser les interactions entre les composants d'un système de la même façon qu'il est indispensable d'isoler les processus les uns des autres à plus bas niveau. On peut supposer que le système comporte un mécanisme de contrôle d'accès qui interdit par exemple à un composant malveillant d'accéder ou de modifier une zone mémoire appartenant à un autre composant. Un tel mécanisme n'est cependant pas suffisant car il

est basé sur les concepts bas-niveau d'adresses et d'espaces mémoire, mais n'intègre pas les notions du canevas comme les composants ou les interfaces. Cela empêche la définition de politiques de sécurité définissant par exemple les droits de liaisons entre les différents composants puisque la liaison est une notion qui n'existe pas au niveau d'un mécanisme d'isolation mémoire.

Le but d'un canevas sécurisé comme celui que nous avons implanté dans THINK est donc de fournir les outils nécessaires à la mise en œuvre de politiques de sécurité régissant les accès entre les composants. Cela passe notamment par l'introduction de mécanismes d'identification des composants qui sont bien évidemment indispensables pour permettre la définition et l'application des droits d'accès de chaque composant par rapport aux autres. De plus, il est nécessaire de fournir les primitives permettant l'application de la politique de sécurité et de les intégrer au cœur même du canevas afin de garantir qu'un composant malveillant ne pourra les court-circuiter. Enfin, dans notre cas particulier qui consiste à sécuriser un canevas existant, il est indispensable de garantir la compatibilité du canevas sécurisé avec le modèle original afin de préserver ses caractéristiques, comme la flexibilité dans le cas du canevas logiciel de THINK.

5.2.2 Etat actuel

Le canevas logiciel de THINK est basé sur les concepts ci-dessous, dont nous détaillons l'implantation dans le prototype pour PowerPC.

5.2.2.1 Le courtier

Le courtier ne fait pas à proprement parler partie du canevas logiciel de THINK. Cependant, sa présence dans un système est fortement recommandée car il permet aux composants de récupérer des noms THINK à partir de noms symboliques (en pratique, des chaînes de caractères) qui sont beaucoup plus simples à manipuler pour des composants clients que des noms THINK. En tant que tel, le courtier ne pose pas de problème de sécurité et nous ne le modifierons pas si ce n'est pour prendre en compte les modifications apportées à l'interface `Name`.

5.2.2.2 Les interfaces

Les interfaces sont implantées dans THINK sous la forme d'une structure comportant deux champs, le premier étant un pointeur sur une autre structure elle-même composée de pointeurs vers les différentes méthodes de l'interface, et le deuxième un pointeur vers les variables d'instances du composant qui sont partagées par toutes ses interfaces. Ce mécanisme est détaillé dans la figure 5.1 sur l'exemple d'un composant exportant deux interfaces `ItfA` et `ItfB` comportant respectivement deux et trois méthodes.

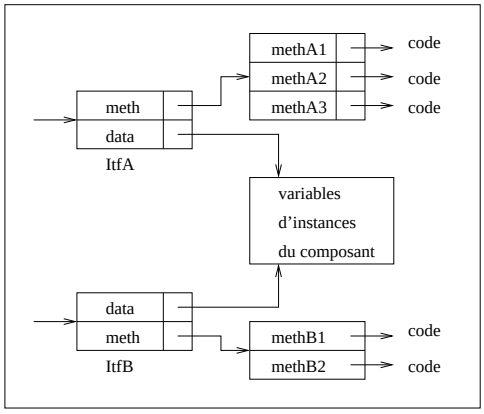

FIG. 5.1 – Implantation des interfaces dans THINK

Du point de vue de la sécurité, cette implantation pose un problème de contrôle d'accès aux méthodes et données. En effet, il est tout à fait normal que quiconque puisse accéder en lecture aux données et en exécution aux méthodes exportées par une interface (puisque c'est justement le rôle d'une interface), mais par contre il n'est pas acceptable qu'un composant puisse modifier l'adresse d'une méthode dans une interface exportée. Il pourrait par exemple faire cela pour rediriger tous les appels à une méthode critique vers une de ses propres méthodes afin de récupérer les paramètres, éventuellement confidentiels, qui sont passés à la méthode initiale. Il serait possible d'interdire l'accès en écriture grâce à un outil d'isolation, matérielle ou logicielle, mais vu le nombre d'interfaces vraisemblablement exportées dans un système, il parait irréaliste de mettre en œuvre une telle isolation pour chaque interface.

5.2.2.3 L'interface `Name`

Les noms dans THINK sont implantés sous la forme d'une structure comprenant deux champs. Le premier est un pointeur vers l'interface `Name` elle-même (c'est à dire l'interface comprenant les méthodes `getDefaultNC` et `toByte`) et le deuxième est un pointeur vers l'interface désignée par ce nom. Cette implantation est détaillée dans la figure 5.2 sur l'exemple d'un composant exportant une interface `ItfA` comprenant trois méthodes.

Cette implantation pose un problème fondamental pour la sécurité du canevas logiciel. En effet, le fait que le nom d'une interface comprenne un pointeur vers cette interface permet à un composant d'appeler les méthodes de cette interface sans créer de liaison avec elle. Dans le modèle de programmation proposé par THINK, un composant doit en effet créer une liaison avec l'interface destination avant de pouvoir appeler ses méthodes. La présence d'un tel pointeur dans la structure du nom THINK peut se justifier pour augmenter les performances du systèmes (puisque la méthode `bind` de l'interface `Binding`

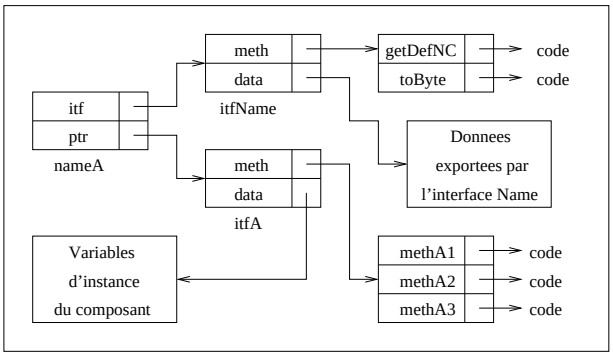

FIG. 5.2 – Implantation de l'interface `Name` dans THINK

`Factory` peut alors être implantée de façon très efficace puisqu'elle n'a qu'à renvoyer le pointeur contenu dans la structure `Name` passée en argument), mais elle paraît inacceptable sur le plan de la sécurité puisqu'elle permet de court-circuiter le mécanisme de liaison.

5.2.2.4 L'interface `Naming Context`

L'interface `Naming Context` comprend deux méthodes, la méthode `byteToName` qui permet d'obtenir un nom à partir de sa forme sérialisées, et la méthode `export` qui permet d'exporter une interface et d'obtenir le nom THINK correspondant. Alors que la première méthode ne pose pas de problème du point de vue de la sécurité du système, la méthode `export` nécessite une sécurisation. En effet, il faut pouvoir vérifier qu'un composant exportant une interface a bien le droit d'exporter cette interface. Un composant malveillant pourrait en effet exporter une interface identique à celle exportée par un composant serveur afin d'usurper l'identité de celui-ci du point de vue des composants clients. En effet, lorsqu'un composant client recherche un composant serveur capable de fournir un certain service, il s'adresse au courtier afin d'obtenir le nom THINK de l'interface d'un composant fournissant ce service. S'il existe dans le système plusieurs composants fournissant le service, le composant client précise quel nom il souhaite obtenir. Dans le prototype de THINK disponible, ce choix est exprimé par le client en fournissant au courtier un indice i désignant le i^{e}composant enregistré. Cette façon de choisir un composant serveur laisse bien entendu la porte ouverte aux abus car il suffirait alors à un composant malveillant de s'enregistrer avant le composant serveur légitime pour être sélectionné si le composant client décide de choisir le 1^{er}composant enregistré ce qui est le choix par défaut. Il serait possible d'interdire à un composant de s'enregistrer dans le courtier sous un nom symbolique donné, mais cela n'empêcherait pas le composant d'exporter tout de même son interface et de diffuser son nom THINK d'une autre manière (un composant doit exporter son interface avant de pouvoir s'enregistrer dans le courtier car il a besoin d'obtenir un

nom THINK pour s'enregistrer).

5.2.2.5 L'interface `Binding Factory`

L'interface `Binding Factory` comprend la méthode `bind` qui permet de créer une liaison entre un composant et une interface. Cette méthode doit être sécurisée de façon à vérifier qu'un composant a bien le droit de se lier avec l'interface en question. En effet, même si on peut vérifier lors de l'appel qu'un composant a bien le droit d'appeler une méthode donnée, le fait de pouvoir récupérer l'interface d'un composant permet d'avoir accès aux données exportés par celui-ci, et il peut être nécessaire de restreindre l'accès à certains composants. De plus, créer une liaison avec une interface peut être une opération au coût non négligeable et consommant des ressources systèmes, par exemple s'il s'agit d'une liaison distante, et en autorisant toutes les créations de liaisons, on expose le système à des attaques par déni de service.

5.2.2.6 Appel de méthodes

L'appel de méthode est matérialisée dans THINK par une « macro » cachant l'appel direct à la méthode via le pointeur stocké dans son interface. La figure 5.3 détaille le code de cette macro.

```
// Definition de la macro
#define CALL(itf, proc, args...) \
  (itf)->meth->proc((itf)->data, ##args)

// Exemple d'utilisation
CALL(allocator, malloc, 1234);

// Developpement de la macro par le preprocesseur
(allocator)->meth->alloc(allocator->data, 1234);
```

FIG. 5.3 – Macro C d'appel à une méthode

Cette façon d'implanter les appels de méthodes inter-composants est insuffisante sur le plan de la sécurité car elle ne permet pas d'effectuer de vérification avant l'appel de méthode. En effet, on peut vouloir vérifier qu'un composant a bien le droit d'appeler telle méthode avant de le laisser effectuer l'appel. Il serait possible en pratique de rajouter une vérification dans la macro (en utilisant l'opérateur virgule du langage C) mais rien n'empêcherait le composant malveillant de ne tout simplement pas utiliser la macro. Le problème vient donc du fait qu'on effectue en fait un appel direct à la méthode en question, même si cela est camouflé par la macro. De plus, cette façon d'appeler une méthode ne fonctionne que si la mémoire est gérée sans isolation matérielle ou du moins si le composant destination se situe dans le même espace d'adressage que le composant source.

En effet, dans le cas contraire, l'appel sera bloqué par le mécanisme d'isolation. Puisque le programmeur système a tout à fait le droit d'utiliser le mécanisme d'isolation matérielle du processeur s'il le souhaite, on doit donc fournir un mécanisme d'appel inter-composant proche des mécanismes de communication inter-processus comme c'est le cas dans les micro-noyaux par exemple.

5.2.3 Proposition d'un canevas sécurisé

Pour pallier les manquements de l'implantation originale du canevas de THINK en matière de sécurité, nous avons défini un canevas sécurisé implantant les mêmes notions que celui présenté dans [Fas01]. Nous avons détaillé ci-dessus les principaux problèmes techniques de l'implantation originale de THINK. On peut ajouter à ceux-ci un problème conceptuel résidant dans le fait qu'il n'existe aucun moyen de désigner un composant dans le canevas original. En effet, les noms THINK représentent des interfaces et non les composants qui les implantent, ce qui empêche la définition de politiques de sécurité précisant les droits d'accès des composants les uns par rapport aux autres. Nous proposons donc un mécanisme d'identification des composants basé sur l'association d'un nom unique défini statiquement pour chaque composant et d'un jeton d'utilisation du canevas alloué dynamiquement et réputé impossible à fabriquer par un composant malveillant. Ce mécanisme ainsi que notre implantation sécurisé du canevas de THINK sont détaillés ci-dessous.

5.2.3.1 Les interfaces

Le modèle d'interface que nous avons défini est détaillé dans la figure 5.4.

```
typedef struct {
  char *staticName;
  unsigned int nMeth;
  struct {
    Meth meth;
    char *methStaticName;
  } *meths;
  unsigned int nData;
  struct {
    Data data;
    char *dataStaticName;
  } *datas;
} Itf;
```

FIG. 5.4 – Nouvelle implantation de la notion d'interface

Le champ `staticName` représente le « nom statique » du composant exportant l'interface. Ce nom statique permet d'identifier de façon unique le composant dans le système,

ce qui n'est pas possible dans l'implantation actuelle de THINK puisque seules les interfaces ont des noms et que ces noms sont dépendants d'un contexte de nommage. Une telle identification unique est nécessaire afin de pouvoir préciser les droits des composants. De plus, il est nécessaire d'être capable de désigner statiquement les composants puisque la politique de sécurité peut être elle-même être écrite statiquement. Le nom statique du composant est donc présent dans toutes les interfaces exportées par ce composant. L'identification d'une interface de façon globale se fait quant à elle grâce au tuple <nom statique, ensemble des méthodes, ensemble des données>.

Le nom statique est déclaré par le composant lorsqu'il s'initialise, et il suffit donc à ce moment de vérifier que le nom déclaré n'est pas déjà enregistré dans le système pour s'assurer de son unicité. En cas de double déclaration, la politique de sécurité doit préciser la réaction (notamment si l'on doit refuser l'enregistrement du deuxième composant seulement ou bien neutraliser aussi le premier, afin d'éviter qu'un composant malveillant vole le nom statique d'un autre). Ce nom statique peut prendre n'importe quelle forme, par exemple l'URL du fichier binaire du composant ce qui garantit son unicité pour les composants liés statiquement au noyau. Cette méthode d'identification assez simpliste est suffisante pour nos besoin dans le cadre de ce travail.

Le champ `nMeth` est le nombre de méthodes exportées dans cette interface. Les méthodes sont décrites dans le champ `meths`, lui-même composé d'un identifiant de méthode (le champ `meth`) et du nom statique de la méthode. L'identifiant est par convention l'adresse mémoire de la méthode, ce qui dans un système gérant la mémoire de façon « plate » (c'est à dire sans segmentation) permet d'optimiser l'appel à la méthode comme ce sera détaillé ci-après. Dans un système utilisant de la mémoire segmentée, et a fortiori dans un système distribué, cet identifiant sert simplement à désigner de manière unique une méthode dans un espace d'adressage donné. Le nom statique de la méthode sert encore une fois à l'écriture de la politique de sécurité et prend la forme du nom de la méthode tel que défini dans le programme du composant.

De façon similaire, les champs `nData` et `data` désignent respectivement le nombre de variables exportées dans l'interface et les variables en question.

5.2.3.2 L'interface `Name`

La figure 5.5 présente la nouvelle implantation de l'interface `Name`.

```
typedef struct {
  unsigned int id;
  NamingContext nc;
} Name;
```

FIG. 5.5 – Nouvelle implantation de l'interface `Name`

L'interface `Name` est composée d'un champ `id` qui permet d'identifier le nom de façon unique dans le contexte de nommage `nc` dans lequel il est défini. Un nom est donc identifié de façon unique dans le système par le tuple <id, nc>. Le champ `id` est simplement un entier sur 32 bits choisit aléatoirement lors de la création du nom.

5.2.3.3 Le gestionnaire de sécurité

Le gestionnaire de sécurité est le composant chargé de mettre en œuvre la politique de sécurité choisie par le programmeur système. Ce composant est un ajout par rapport au canevas original et a pour but de centraliser la gestion de la sécurité (bien qu'en pratique, ce gestionnaire peut tout à fait être séparé en plusieurs composants afin d'améliorer sa capacité à passer à l'échelle et éviter les goulots d'étranglement). L'interface du gestionnaire de sécurité est présenté dans la figure 5.6.

```
Secret getSecret(Itf itf);

boolean checkExport(Secret secret,
                    NamingContext nc,
                    Itf itf);

boolean checkBind(char *staticName,
                  Secret secret,
                  Name dstName);

boolean checkCall(char *staticName,
                  Secret secret,
                  Itf dstItf,
                  char *methStaticName);
```

FIG. 5.6 – Interface du gestionnaire de sécurité

La méthode `getSecret` doit être appelée par chaque composant avant d'utiliser les autres méthodes du canevas logiciel sécurisé. Elle permet à un composant d'obtenir un secret, c'est à dire un jeton d'utilisation du canevas. Ce secret permet d'identifier de façon incontestable un composant, de la même façon qu'une clé privée PGP [Zim95] permet d'identifier un utilisateur (bien qu'il ne soit pas question ici de cryptologie). Ce secret est implanté dans notre prototype comme un nombre aléatoire sur 128 bits, ce qui rend statistiquement très peu probable la possibilité pour un composant de fabriquer lui-même un tel secret. A noter que le secret n'est pas dépendant de l'interface `itf` passée en argument, et deux appels à la méthode `getSecret` avec la même interface en argument ne renverront pas le même secret. Le passage de cette interface a seulement pour but de permettre d'associer le secret généré au nom statique du composant exportant l'interface. Au cas où l'on souhaiterait construire un système très sécurisé, il est possible de remplacer ce secret

simpliste par un jeton plus sécurisé, par exemple une clé PGP sur 4096 bits. Cette méthode d'authentification permet d'assurer un niveau de sécurité suffisant pour nos prototypes.

La méthode `checkExport` a pour but de vérifier que le composant identifié par son nom statique (compris dans l'interface exportée) et son secret a bien le droit d'exporter l'interface `itf` dans le contexte de nommage `nc`. Après avoir vérifié que le secret passé en argument est bien un secret généré par le gestionnaire de sécurité et qu'il correspond bien au nom statique de l'interface exportée, la méthode vérifie la validité de l'exportation. Cette vérification est double. Tout d'abord le gestionnaire de sécurité vérifie que cette interface n'est pas déjà exportée dans le contexte de nommage. En dehors d'une erreur de programmation, une double déclaration peut être le symptôme d'une tentative d'usurpation d'identité. En effet, un composant peut essayer d'exporter une interface identique à celle d'un composant serveur afin de tromper les composants clients qui souhaitent y faire appel. Si le composant malveillant exporte l'interface litigieuse avant le composant serveur légitime, le gestionnaire de sécurité n'a pas de moyen de l'en empêcher, mais lorsque le composant légitime exportera son interface à son tour, le gestionnaire de sécurité détectera la double déclaration et pourra alors agir contre l'un ou l'autre (ou les deux) composants, selon ce qu'il a été décidé dans la politique de sécurité. La méthode `checkExport` vérifie ensuite que le composant désigné par son secret a bien le droit d'exporter l'ensemble de méthodes et de données composant l'interface exportée. Cela fournit une sécurité supplémentaire contre les composants cherchant à usurper l'identité d'un autre composant en exportant une interface identique.

La méthode `checkBind` vérifie que le composant identifié par son nom statique et son secret a bien le droit de se lier à l'interface identifié par son nom THINK `dstName`. Cette méthode vérifie donc que le secret passé est bien valide et correspond bien au composant dont le nom statique est passé en argument, puis que la création de la liaison est autorisée par la politique de sécurité.

La méthode `checkCall` est utilisée lors de l'appel inter-composant. Son rôle est de vérifier que l'appel est bien autorisé par la politique de sécurité. Après avoir vérifié que le secret passé correspond bien au composant identifié par son nom statique, la méthode s'assure que la politique de sécurité autorise bien ce composant à appeler la méthode `methStaticName` de l'interface `dstItf`.

5.2.3.4 L'interface `Naming Context`

Le composant gérant les contextes de nommage contient une base stockant les associations <nom, interface>. Cette base est présentée dans la figure 5.7.

Cette implantation est équivalente à celle réalisée dans le prototype original de THINK et ne pose pas de problème de sécurité.

Les nouveaux prototypes des fonctions implantant l'interface `Naming Context` sont

```
typedef struct {
  unsigned int n;
  struct {
    Name name;
    Itf itf;
  } *tuples;
} NamingContextData;

static NamingContextData *namingContexts;
```

FIG. 5.7 – Nouvelle implantation de la base stockant les contextes de nommage

détaillés dans la figure 5.8.

```
Name byteToName(NamingContext nc,
                char *str);

Name export(NamingContext nc,
            Secret secret,
            Itf itf);
```

FIG. 5.8 – Nouvelle implantation de l'interface `Naming Context`

Le paramètre `nc` des deux méthodes est la traduction C de la construction syntaxique Java `nc.byteToName` et `nc.export` et n'a donc pas de rôle sémantique.

La méthode `export` a un argument de plus que la méthode implantée dans le prototype initial de THINK. Le secret sert à identifier le composant appelant (ainsi que le nom statique qui est compris dans l'interface exportée). Il n'est pas utilisé par la méthode `export` elle-même, mais par la méthode `checkExport` qui est appelée avant toute chose pour vérifier que l'opération est autorisée. L'exportation quand à elle consiste simplement à créer un nom THINK et à enregistrer l'association entre ce nom et l'interface exportée dans le contexte de nommage.

5.2.3.5 L'interface `Binding factory`

L'interface `Binding factory` comporte une seule méthode, `bind`, dont le prototype est détaillé dans la figure 5.9.

```
Itf bind(char *staticName,
         Secret secret,
         Name dstName);
```

FIG. 5.9 – Nouvelle implantation de l'interface `Binding Factory`

Le prototype de la méthode `bind` diffère de celui implanté dans la version originale de

THINK. En plus du nom THINK de l'interface avec laquelle on veut se lier, `dstName`, le composant source passe en argument son nom statique et son secret pour s'identifier. La méthode `bind` commence donc par vérifier que l'interface de nom THINK `dstName` avec laquelle le composant veut se lier existe bien puis appelle la méthode `checkBind` pour s'assurer que le composant source a bien le droit de se lier avec cette interface. Finalement, la liaison en tant que telle revient simplement à renvoyer une copie de l'interface associée au nom THINK `dstName` dans le contexte de nommage où il est défini. Le fait de renvoyer une copie de l'interface plutôt qu'un pointeur vers celle-ci permet d'éviter qu'un composant malveillant ne modifie une interface de façon globale.

5.2.3.6 Appel de méthodes

Pour pallier les insuffisances de la procédure d'appel de méthodes implantée dans le prototype original de THINK, nous avons défini une interface, nommée `Inter-Component Communication` et présentée dans la figure 5.10.

```
void *call(char *staticName,
           Secret secret,
           Itf dstItf,
           char *methStaticName,
           void *args);
```

FIG. 5.10 – Implantation de l'interface `Inter-Component Communication`

La méthode `call` de l'interface `Inter-Component Communication` prend comme paramètre le nom statique et le secret du composant source afin de pouvoir l'identifier, ainsi que l'interface `dstItf` contenant la méthode `methStaticName` à appeler, avec les paramètres stockés dans le tableau `args`. L'identification du composant source est utile à la méthode `checkCall` qui est appelée afin de vérifier que l'appel est bien autorisé par la politique de sécurité. Une fois la vérification effectuée, l'appel est réalisé en fonction de l'isolation mise en place entre les composants.

Dans le cas le plus simple, il n'y a pas d'isolation entre les deux composants (par exemple s'ils sont dans le même segment mémoire) et l'appel peut être réalisé directement en récupérant l'adresse de la méthode dans l'interface `dstItf`. A noter que dans ce cas bien sûr, rien n'empêche le composant source de court-circuiter le mécanisme de vérification des autorisation de liaison et d'appel en appelant directement la méthode grâce à son adresse.

Dans un cas plus réaliste, les deux composants sont isolés et le composant source est obligé de passer par la méthode `call` pour appeler les méthodes du composant destination. La méthode `call` a donc pour rôle de relayer l'appel que le composant source ne peut effectuer directement. Cela implique que le composant implantant la méthode `call` doit avoir le droit d'exécution dans les zones mémoire contenant les composants applicatifs,

ce qui n'est pas déraisonnable si on considère que ce composant fait partie du canevas sécurisé et peut donc être légitimement privilégié par rapport aux applications. Dans le cas d'une isolation matérielle basée sur la segmentation de la mémoire, le segment contenant le composant destination devra interdire les appels depuis le segment contenant le composant source et autoriser ceux provenant du segment contenant la méthode `call`. Par exemple, le processus Pentium d'Intel [Int97] permet la mise en place de ce type d'architecture grâce à la segmentation de la mémoire dont la sécurité est basée sur un mécanisme simple de niveau de privilège, et à un mécanisme d'appel inter-segment appelé portails d'appel (de l'anglais *call gates*). Dans le cas d'une isolation logicielle basée sur de l'injection de code de vérification, comme effectué par notre outil élémentaire d'isolation mémoire, il suffit de générer du code qui interdira les appels entre les deux composants applicatifs mais autorisera les appels vers et depuis le composant implantant la méthode `call`.

Enfin, dans le cas où les deux composants sont distribués sur des machines différentes, l'appel prendra la forme d'un appel de procédure distante (de l'anglais *Remote Procedure Call*) comme le protocole présenté dans [BALL89] par exemple.

5.3 Expression de la politique de sécurité

La politique de sécurité regroupe l'ensemble des droits des différents composants les uns par rapport aux autres. Dans le cas de la protection, cette politique est en fait simplement une expression de la matrice de Lampson qui définit les droits d'accès des différents composants, auxquels on ajoute les droits de modification de la matrice elle-même par les composants. Toute la difficulté se situe alors dans la façon d'exprimer cette politique en conciliant les différents objectifs.

5.3.1 Problématique

L'expression de la politique de sécurité est un problème complexe. L'objectif est triple : on doit fournir au programmeur système un moyen de spécifier sa politique de sécurité de façon simple et portable, on doit garantir la flexibilité de la gestion de la politique afin de permettre son évolution dynamique pendant l'exécution du système, et on doit s'assurer que l'analyse et la mise en œuvre de la politique seront effectuées de façon suffisamment performante pour ne pas pénaliser l'exécution du système.

Bien que de nombreux travaux aient été conduit concernant la gestion des politiques de protection au sens large (contrôle d'accès, gestion de ressources, etc) dans les systèmes distribués [Slo94] [LMSY95] [MS93], la plupart des résultats obtenus sont difficilement applicables dans un noyau de système d'exploitation. En effet, les performances sont essentielles dans un noyau et la plupart des systèmes de gestion flexible des politiques de sécurité se basent sur l'analyse dynamique de contraintes, souvent exprimées dans un

langage adapté à la résolution de contraintes comme Prolog [DEDC96] par exemple.

Bien entendu, si Prolog se révèle être un langage tout à fait adapté à la spécification de politique de sécurité du point de vue de la facilité de programmation et de la flexibilité, il en est autrement du point de vue des performances et on imagine mal un système d'exploitation émettant une requête Prolog à chaque appel de méthode pour vérifier si l'appel est autorisé. A contrario, une politique codée directement en C dans le composant gestionnaire de sécurité présente l'avantage d'être très optimisée du point de vue des performances, mais par contre se révèle beaucoup plus fastidieuse à spécifier puisque le programmeur système doit prendre en compte la façon dont a été implanté le gestionnaire de sécurité, ce qui de plus rend la politique difficilement portable d'un système à un autre.

On présente ici des propositions pour la gestion de politiques de sécurité dans une architecture de systèmes flexibles telle que THINK. Nous détaillons tout d'abord une proposition pour la spécification statique de politiques, avant de présenter des pistes concernant la modification dynamique d'une politique de sécurité existante.

5.3.2 Proposition pour la spécification statique de politiques

Nous proposons de concilier la simplicité de spécification et la portabilité du langage Prolog avec les performances d'une implantation en C. Pour cela, on fournit un outil capable de générer le code C permettant le chargement dans le gestionnaire de sécurité d'une politique de sécurité définie statiquement en utilisant un sous-ensemble du langage Prolog. On détaille le fonctionnement de cet outil sur un exemple simple.

On prend l'exemple de deux composants C1 et C2, exportant chacun une interface. Ces interfaces sont présentées dans la figure 5.11.

```
interface itfC1 {
  void m11();
  void m12();
}

interface itfC2 {
  void m21();
  void m22();
}
```

FIG. 5.11 – Interfaces des composants C1 et C2

Ces deux composants sont isolés grâce à un mécanisme d'isolation mémoire logicielle. La figure 5.12 résume les droits des différents segments contenant les composants tels qu'ils sont mis en œuvre par l'outil d'isolation.

Comme détaillé dans la figure 5.12, les segments contenant les composants C1 et C2 sont complètement isolés l'un de l'autre, ce qui implique que les composants doivent

	SegC1	SegC2	SegCanevas
SegC1	RWX	∅	X
SegC2	∅	RWX	X
SegCanevas	RWX	RWX	RWX

FIG. 5.12 – Droits des différents segments pour l'isolation logicielle

obligatoirement passer par les méthodes du canevas logiciel pour communiquer. Dans cet exemple, les droits d'accès utilisés par les méthodes `checkExport`, `checkBind` et `checkCall` sont gérés sous formes de capacités confinées qui précisent pour chaque composant les droits qui y sont associés. La figure 5.13 détaille la structure de donnée servant au gestionnaire de sécurité pour le stockage des capacités. Cet exemple de structure de donnée est adapté à une gestion simple de capacités liées au seul contrôle des communications inter-composants.

```
typedef struct {
  unsigned int n;
  struct {
    char *staticName;
    Secret secret;
    unsigned int nExportRights;
    struct {
      NamingContext nc;
      Itf itf;
    } *exportRights;
    unsigned int nBindRights;
    struct {
      Itf itf;
    } *bindRights;
    unsigned int nCallRights;
    struct {
      Itf itf;
      char *methStaticName;
    } *callRights;
  } *capa;
} Capabilities;

static Capabilities capas;
```

FIG. 5.13 – Structure de donnée pour la gestion des capacités

Le champ `n` indique le nombre de capacités gérées par le gestionnaire de sécurité. Le champ `capa` stocke les capacités. Il contient, pour chaque capacité, le nom statique (`staticName`) et le secret (`secret`) du composant sujet de la capacité, ainsi que le nombre de capacités d'exportation (`nExportRights`) et ces capacités d'exportation (`exportRights`, le nombre de capacité de liaison (`nBindRights`) et ces capacités de liaison (`bindRights`),

et le nombre de capacités d'appel (`nCallRights`) et ces capacités d'appel (`callRights`). Chaque capacité d'exportation contient une interface (`itf`) que le composant sujet a le droit d'exporter dans le contexte de nommage (`nc`). De façon similaire, chaque capacité de liaison contient une interface avec laquelle le composant sujet a le droit de se lier. Enfin, chaque capacité d'appel contient le nom statique (`methStaticName`) d'une méthode que le composant sujet a le droit d'appeller, ainsi que l'interface dans laquelle cette méthode est exportée.

On souhaite mettre en place une politique qui permette aux composants C1 et C2 d'effectuer les opérations décrites dans la figure 5.14.

	export		bind		call			
	itfC1	itfC2	nameC1	nameC2	C1.m11	C1.m12	C2.m21	C2.m22
C1	oui	non	non	oui	non	non	oui	oui
C2	non	oui	oui	non	non	oui	non	non

FIG. 5.14 – Opérations autorisées et interdites pour les composants C1 et C2

A noter qu'on interdit à un composant de se lier avec lui-même et d'appeler ses propres méthodes via le canevas logiciel. C'est un choix de politique qui peut se justifier pour optimiser le système en forçant un composant à appeler directement ses méthodes.

Le programmeur système spécifie donc statiquement cette politique en utilisant les prédicats présentés dans la figure 5.15. On notera qu'il s'agit de prédicats au sens Prolog à savoir de règles sans partie droite, puisque notre outil ne gère pas les clauses de Horn conditionnelles. Le sous-ensemble de Prolog que nous gérons sert donc uniquement à spécifier statiquement les droits des différents composants.

```
interface(itfC1, "C1", 2, "m11", "m12", 0).
interface(itfC2, "C2", 2, "m21", "m22", 0).

component("C1").
export("C1", itfC1, 0).
bind("C1", itfC2).
call("C1", itfC2, "m21").
call("C1", itfC2, "m22").

component("C2").
export("C2", itfC2, 0).
bind("C2", itfC1).
call("C2", itfC1, "m12").
```

FIG. 5.15 – Spécification de la politique de sécurité en Prolog

Les deux axiomes `interface(itf1, ...)` et `interface(itf2, ...)` permettent de spécifier les interfaces des composants C1 et C2. Ainsi, l'axiome `interface(itf1, "C1",`

`2, "m11", "m12", 0).` spécifie l'interface ItfC1 présentée dans la figure 5.11, et précise le nom statique du composant exportant cette interface (`C1`), le nombre de méthodes exportées (`2`), les noms statiques de ces méthodes (`m11` et `m12`) et le nombre de données exportées (`0`). Les axiomes `component("C1")` et `component("C1")` permettent de spécifier les noms statiques des composants du système. Les axiomes `export(...)` précisent les droits d'exportation des différentes composants. Par exemple, l'axiome `export("C1", itfC1, 0).` spécifie que le composant C1 a le droit d'exporter l'interface itfC1 dans le contexte de nommage 0. Les axiomes `bind(...)` permettent de spécifier les droits de liaison des composants. Ainsi, l'axiome `bind("C1", itfC2).` déclare que le composant C1 a le droit de se lier avec l'interface itfC2. Enfin, les axiomes `call(...)` spécifient les droits d'appel des différents composants. Par exemple, l'axiome `call("C1", itfC2, "m21").` spécifie que le composant C1 a le droit d'appeler la méthode m21 de l'interface itfC2.

En utilisant notre outil, on génère à partir de cette spécification une fonction C dont le rôle est d'initialiser la structure de donnée présentée dans la figure 5.13. Cette fonction est ensuite liée avec le noyau généré et appelé à l'initialisation du gestionnaire de sécurité. Un résumé du code généré est présenté dans la figure 5.16.

On voit donc qu'il est assez simple de fournir au programmeur un outil lui permettant de traduire une politique définie statiquement dans un sous-ensemble du langage Prolog vers un code C adapté à la représentation des listes de capacités gérées par le gestionnaire de sécurité. Cet outil a cependant le désavantage d'être totalement dépendant de l'implantation de la protection, dans notre cas des capacités gérant les droits d'exportation, de liaison et d'appel. Cette dépendance ne nous paraît cependant pas très contraignante car le programmeur système n'aura qu'à modifier la fonction de génération de code de notre outil pour qu'il génère le code C adapté à son implantation de la protection. De plus, cela permet de générer du code optimisé à l'implantation du gestionnaire de sécurité. A contrario, la spécification de la politique est indépendante de l'implantation de la protection et c'est l'outil qui se charge de l'instancier dans le gestionnaire de sécurité. La gestion de la politique de sécurité bénéficie donc de la simplicité de déclaration et de la portabilité d'un langage de haut-niveau comme Prolog, couplé avec les performances du langage C dans lequel le code intégré au système est généré.

5.3.3 Proposition pour la modification dynamique de politiques

La proposition précédente est intéressante pour les politiques spécifiées statiquement, c'est à dire écrites par le programmeur système au moment de la construction de son système. Il est ensuite possible de changer la politique dynamiquement en modifiant directement les capacités stockées dans le gestionnaire de sécurité, mais on aimerait fournir au programmeur le même type d'outil que pour la spécification statique de la politique afin de lui faciliter la tâche. Cependant, si le fait d'inclure l'outil de traduction des règles Prolog

```
void loadPolicy() {
  Itf itfC1 = {"C1", 2, {{"m11", 0}, {"m12", 0}}, 0};
  Itf itfC2 = {"C2", 2, {{"m21", 0}, {"m22", 0}}, 0};
//
  capas.capa[0].staticName = "C1";
  capas.capa[0].exportRights[0].itf = itfC1;
  capas.capa[0].exportRights[0].nc = 0;
  capas.capa[0].bindRights[0].itf = itfC2;
  capas.capa[0].callRights[0].itf = itfC2;
  capas.capa[0].callRights[0].methStaticName = "m21";
  capas.capa[0].callRights[1].itf = itfC2;
  capas.capa[0].callRights[1].methStaticName = "m22";
  capas.capa[0].nExportRights = 1;
  capas.capa[0].nBindRights = 1;
  capas.capa[0].nCallRights = 2;
//
  capas.capa[1].staticName = "C2";
  capas.capa[1].exportRights[0].itf = itfC1;
  capas.capa[1].exportRights[0].nc = 0;
  capas.capa[1].bindRights[0].itf = itfC1;
  capas.capa[1].callRights[0].itf = itfC1;
  capas.capa[1].callRights[0].methStaticName = "m12";
  capas.capa[1].nExportRights = 1;
  capas.capa[1].nBindRights = 1;
  capas.capa[1].nCallRights = 1;
//
  capas.n = 2;
}
```

FIG. 5.16 – Code C généré à partir de la politique Prolog

dans le noyau afin de pouvoir s'en servir pendant l'exécution ne pose pas de problème technique, il convient d'être prudent quand au fait d'autoriser les composants à modifier la politique de sécurité du système.

Ainsi, on peut mettre en place une gestion à deux niveaux de la politique. Tout d'abord au niveau statique on précise quels composants peuvent modifier la politique de sécurité et de quelle façon. Puis au niveau dynamique, on fournit les outils permettant aux composants d'ajouter des règles (et on vérifie qu'un composant a bien le droit d'ajouter la règle en question). Cependant, ce mécanisme simpliste cache en fait une difficulté très importante : comment anticiper les règles qu'un composant pourra vouloir ajouter. Dans un système gérant le chargement dynamique de composants, il est impossible de prévoir quel composant sera ajouté au système et donc son comportement. On risque donc d'être soit trop restrictif, soit trop large. Par exemple, une règle du type `addBind("C1", "C1", itfC3).` autorise le composant C1 a ajouter une règle `bind("C1", itfC3).` qui l'autorise à se lier à l'interface ItfC3. Une telle règle est très restrictive et ne pose pas de problème de sécurité puisqu'on précise exactement la règle que le composant a le droit d'ajouter, mais elle n'a

pas non plus de réel intérêt puisqu'on pourrait statiquement ajouter la règle `bind("C1", itfC3).` pour autoriser le composant à se lier à itfC3 de toute façon, qu'il en ait besoin ou pas lors de l'exécution du système. A contrario, une règle du type `addBind("C1", "C1", _itf).` qui autorise le composant C1 à ajouter des règles l'autorisant à se lier à n'importe quelle interface n'est pas assez restrictive car on peut avoir besoin d'interdire la liaison avec certaines interfaces.

La gestion dynamique de politiques de sécurité est donc un problème particulièrement complexe. Une hypothèse simplificatrice peut cependant être prise pour permettre d'implanter des outils d'ajout et de modification de règles. Ainsi, si on considère que chaque modification de la politique non explicitement autorisée dans la politique elle-même peut être notifiée à un administrateur humain qui prendra la décision finale, on peut mettre en œuvre un exemple de modification dynamique de la politique de sécurité.

On part donc de l'exemple détaillé précédemment avec les composants C1 et C2 qui souhaitent communiquer. L'aspect dynamique est matérialisé par l'ajout d'un composant C3 durant l'exécution du système. La figure 5.17 présente l'interface exportée par le composant C3.

```
interface itfC3 {
  void m31();
  void m32();
}
```

FIG. 5.17 – Interfaces du composant C3

Le composant C3 est isolé des composants C1 et C2 grâce à l'isolation logicielle des trois segments contenant les composants, et il est donc obligé de passer par les méthodes du canevas logiciel pour communiquer. La politique définie statiquement pour préciser les droits de communication entre les composants est la même que dans la figure 5.15, puisqu'on ne sait pas au moment de la compilation quel composant va être ajouté dynamiquement et qu'on ne peut donc pas préciser les droits associés au composant C3. Par contre, on doit ajouter les règles décrites dans la figure 5.18 pour permettre au composant C3 de s'enregistrer dans le gestionnaire de sécurité lorsqu'il sera chargé.

```
addComponent(_c).
addInterface(_i, _c, _nm, _ml, _nd, _dl).
```

FIG. 5.18 – Spécification statique des droits d'ajout d'un composant

La règle `addComponent(_c).` précise que de nouveaux composants peuvent être ajoutés dynamiquement au système. La règle `addInterface(...).` spécifie quand à elle qu'il est possible pour de nouveaux composants ajoutés au système de définir leurs interfaces. Le

code C généré pour ces deux règles est présenté dans la figure 5.19.

```
capas.addItf = true;
capas.addComp = true;
```

FIG. 5.19 – Code C généré à partir des nouvelles règles Prolog

Le code généré a donc simplement pour effet d'affecter deux booléens stockés dans la structure présentée dans la figure 5.13, modifiée pour ajouter les deux champs en question.

A l'exécution, le composant C3 va donc s'initialiser lors de son chargement et demander l'insertion de la règle de définition d'une interface `interface(itfC3, "C3", 2, "m31", "m32", 0).` et de la règle de définition d'un composant `component("C3").` afin de s'enregistrer auprès du gestionnaire de sécurité. Pour cela, le gestionnaire de sécurité fournit une méthode `loadRule(char *rule)` dont le rôle est d'interpréter la règle Prolog fournie sous forme de chaîne de caractère pour initialiser correctement la structure présentée dans la figure 5.13. A la différence de l'outil utilisé pour la génération de code C basé sur la spécification statique d'une politique de sécurité, la méthode `loadRule` ne génère pas de code mais effectue elle-même le traitement associé à l'application de la règle en question. La méthode `loadRule` vérifie donc que l'ajout de composant et d'interface est bien autorisé par la politique (en consultant les booléens `addComp` et `addItf` ajoutés à la structure de la figure 5.13) avant d'effectuer la modification de la structure en question en ajoutant une nouvelle capacité. Cette action d'ajout est décrite dans la figure 5.20 sous la forme du code C exécuté par la méthode `loadRule`.

```
capas.n ++;
capas.capa[2].staticName = "C3";
```

FIG. 5.20 – Code exécuté par la méthode `loadRule` pour l'enregistrement du composant C3

Par contre, lorsque le composant C3 va appeler la méthode `export` de l'interface `Naming Context` pour exporter son interface itfC3, la méthode `checkExport` du gestionnaire de sécurité ne trouvera pas dans la structure contenant les capacités l'autorisation d'exportation pour C3. Cette méthode devra alors faire appel à l'administrateur qui décidera si le composant a bien le droit d'exporter l'interface en question et le cas échéant, exécutera lui-même la méthode `loadRule("export("C3", itfC3, 0).")` pour ajouter dans la structure décrite dans la figure 5.13 le droit d'exportation de l'interface itfC3 par le composant C3. Le code exécuté pour cela est présenté dans la figure 5.21.

Le processus est similaire pour l'ajout de droits de liaison et d'appel. Cet exemple simple montre qu'il est possible de modifier dynamiquement une politique de sécurité en bénéficiant des avantages d'un langage haut-niveau comme Prolog. Il est bien sûr tout à fait

```
capas.capa[2].exportRights[0].itf = itfC3;
capas.capa[2].exportRights[0].nc = 0;
capas.capa[2].nExportRights = 1;
```

FIG. 5.21 – Code exécuté par la méthode `loadRule` pour l'ajout du droit d'exportation de itfC3

possible de se passer d'un tel langage et d'effectuer directement les modifications voulues dans la structure gérant les capacités, mais le composant effectuant la modification est alors dépendant de l'implantation de cette structure et plus généralement de la façon dont sont gérés les droits d'accès dans le système. La méthode `loadRule` permet de s'abstraire de cette implantation en utilisant un sous-ensemble du langage Prolog pour spécifier les droits. Néanmoins, l'outil de traduction que nous proposons ne permet pas la définition de politiques quelconque. En effet, la traduction d'un règle par la méthode `loadRule` suppose que cette règle a été prévue par le programmeur de la méthode. Il n'est donc pas possible d'ajouter de nouvelles règles quelconques comme c'est le cas en Prolog.

5.4 Conclusion

Comme nous l'avons vu dans ce chapitre, il est possible de mettre en place un canevas logiciel sécurisé qui préserve la flexibilité du système. La sécurisation du canevas logiciel de THINK a nécessité une réimplantation complète de celui-ci puisque nous avons modifié ses types de bases, les interfaces et les noms. Cela illustre d'ailleurs le fait qu'il est toujours complexe de sécuriser a posteriori un système (ou dans le cas présent un canevas logiciel) qui n'a pas été prévu pour cela initialement car cela entraîne en général beaucoup de modifications dans le reste du code. Au final, le canevas sécurisé que nous proposons est conforme au modèle de programmation de l'architecture THINK ce qui garantit sa flexibilité. Ces résultats peuvent être généralisés au-delà de THINK vers tous les systèmes proposant un modèle de programmation uniforme qui impose un minimum de règles lors du développement de modules noyaux, en particulier en ce qui concerne les appels de méthodes.

Les méthodes de vérification des droits d'exportation, de liaison et d'appel permettent d'assurer un contrôle de façon transparente pour les composants applicatifs. Ces méthodes fournies par le gestionnaire de sécurité consultent la politique de sécurité pour accorder ou non le droit demandé. La gestion de la politique est donc entièrement assurée dans le gestionnaire de sécurité, conformément à notre volonté de dissocier la gestion de la politique des outils servant à sa mise en œuvre.

Concernant la gestion de la politique de sécurité, l'outil que nous proposons permet de spécifier des politiques simples en utilisant un sous-ensemble du langage Prolog et de les

traduire en C. Bien évidemment, cet outil est complètement dépendant de notre exemple d'implantation de la protection dans le canevas, mais il illustre comment il est possible de fournir des outils d'aide à la spécification de politiques statiques. Concernant la modification dynamique des politiques de sécurité, nous avons vu qu'elle était possible à condition que les règles ajoutées aient été prévus par le programmeur du gestionnaire de sécurité, et en laissant au gestionnaire de sécurité la possibilité de consulter l'administrateur du système pour les cas où la règle n'a pu être prévu statiquement. Nous présentons dans le chapitre suivant différentes applications de nos résultats afin d'évaluer la flexibilité de la protection offerte par les outils élementaires et le canevas sécurisé.

Chapitre 6

Applications et évaluation

6.1 Introduction

Nous présentons dans ce chapitre différentes applications de nos résultats. Nous démontrons tout d'abord que nos outils sont indépendants des politiques de sécurité mises en œuvre en montrant que l'on peut faire évoluer ces politiques sans avoir à modifier les outils. Nous détaillons ensuite un canevas partiel de gestion de ressources que nous avons implanté dans l'architecture THINK, afin de valider notre modèle de canevas sécurisé. Enfin nous montrons comment le canevas et les outils peuvent être utilisés dans un exemple concret, la construction d'un environnement d'exécution sécurisé pour routeurs actifs. Tous ces exemples ont pour but de démontrer que notre approche de séparation de la gestion de la politique de sécurité des outils permettant sa mise en œuvre garantit la flexibilité de la protection, tout en maintenant un niveau de sécurité suffisant pour des applications réalistes comme le montre l'exemple des réseaux actifs.

6.2 Modification de la politique d'isolation mémoire

6.2.1 Utilisation de l'outil d'isolation logicielle

Pour illustrer la flexibilité de notre mécanisme d'isolation mémoire, nous détaillons un exemple d'utilisation où la politique d'isolation entre deux composants est modifiée dynamiquement. Supposons par exemple que le composant C1 souhaite accéder à un ensemble de variables partagées par le composant C2 qui vient d'être chargé dynamiquement dans le système. Afin de s'assurer que le composant C1 ne pourra accéder à ce composant avant qu'il ne soit complètement initialisé, on décide que C2 est initialement complètement isolé de C1, comme exprimé par la politique décrite dans la figure 6.1.

Cette isolation est réalisée au niveau système, grâce à l'outil d'isolation mémoire logicielle. La matrice d'accès détaillant les droits d'accès des segments contenant les compo-

```
read(SegC1, SegC1).
write(SegC1, SegC1).
exec(SegC1, SegC1).

read(SegC2, SegC2).
write(SegC2, SegC2).
exec(SegC2, SegC2).
```

FIG. 6.1 – Spécification de la politique d'accès entre les segments contenant C1 et C2

sants C1 et C2 est présentée dans la figure 6.2.

	SegC1	SegC2
SegC1	RWX	∅
SegC2	∅	RWX

FIG. 6.2 – Matrice d'accès pour les segments contenant C1 et C2

Ces droits d'accès sont stockés dans le gestionnaire de sécurité sous formes de capacités de façon similaire à la gestion des droits d'accès dans le canevas sécurisé. En pratique, ces droits sont stockés dans un tableau de booléens indexé par le numéro du segment, comme présenté dans la figure 6.3.

```
typedef struct {
  unsigned n;
  struct {
    unsigned seg;
    int read:1;
    int write:1;
    int exec:1;
  } *capa;
} Capacities;

Capacities *capas;
```

FIG. 6.3 – Tableau des capacités d'accès des différents segments

Le tableau `capas` contient pour chaque segment S une structure de type `Capacities`. Le champs `n` de cette structure indique le nombre de segments pour lesquels le segment S a des capacités d'accès. Le champs `capa` est un tableau contenant les capacités du segment. Chaque capacité est elle-même une structure contenant 4 champs. Le champs `seg` est le numéro du segment destination. Les champs `read`, `write` et `exec` sont des booléens valant vrai si et seulement si le segment S a le droit respectivement de lire, écrire ou exécuter à une adresse appartenant au segment destination.

Comme détaillé précédemment, l'isolation est réalisée en remplaçant tous les accès

mémoire par un branchement vers le gestionnaire de sécurité, qui vérifie si l'accès est autorisé. Le code de vérification généré dynamiquement accède au tableau `capas` pour rechercher s'il existe une capacité autorisant l'accès mémoire. Pour cela, le gestionnaire de sécurité doit connaître le numéro du segment contenant le composant appelant, le numéro du segment contenant la zone mémoire accédée, et le type d'accès (c'est à dire une lecture, une écriture ou un branchement). Le numéro du segment source peut être obtenu grâce à l'adresse de l'instruction remplacée. Lors de l'appel au gestionnaire de sécurité, on utilise une instruction d'appel de procédure (`call`) qui stocke l'adresse de retour dans un registre réservé à cet effet ou sur la pile selon les architectures. Le numéro du segment est typiquement représenté par les bits de poids forts des adresses et peut donc facilement être extrait de cette adresse. De la même façon, l'adresse destination est calculée par le code généré pour l'implantation de la correspondance de segments logicielle et on peut donc en extraire le numéro de segment de destination. Enfin, le type de l'instruction est connu lors de la génération du code qui est donc généré de façon à rechercher la bonne capacité.

Une fois que le composant C2 est initialisé, on souhaite donner au composant C1 accès au segment le contenant. Pour cela, il suffit de modifier le champs `read` de la capacité décrivant les droits d'accès du segment contenant C1 vers le segment contenant C2. En supposant que l'on dispose de notre outil d'insertion dynamique de règles, il suffit d'appeler la méthode `loadRule("read(SegC1, SegC2).")` pour accorder au composant C1 le droit de lire à l'intérieur du segment contenant C2. La modification est immédiatement prise en compte et ne nécessite notamment pas de re-générer le code de vérification qui ne fait que rechercher la capacité nécessaire. Comme on le voit, la modification dynamique d'une politique d'isolation est extrêmement simple grâce à la séparation entre l'outil servant à mettre en œuvre la politique (i.e : le mécanisme d'isolation mémoire logicielle) et la gestion de cette politique dans le gestionnaire de sécurité. La flexibilité de la politique d'isolation est donc garantie.

6.2.2 Combinaison de l'outil d'isolation logicielle et du canevas sécurisé

La figure 6.4 illustre une architecture d'isolation pouvant être mise en place dans un système réaliste et regroupant l'isolation mémoire logicielle assurée par l'outil d'isolation, et l'isolation des composants réalisée grâce au canevas sécurisé. Cette architecture d'isolation est illustrée sur un exemple simple où le composant C1 désire appeler la méthode `m2` du composant C2.

Les traits en pointillés délimitent les segments tels que définis par l'outil d'isolation logicielle. Ces segments assurent que les composants ne pourront pas court-circuiter le canevas logiciel pour ce qui concerne les appels de méthodes, mais autorise par contre l'accès en lecture aux variables partagées du composant C2. La politique d'isolation initiale

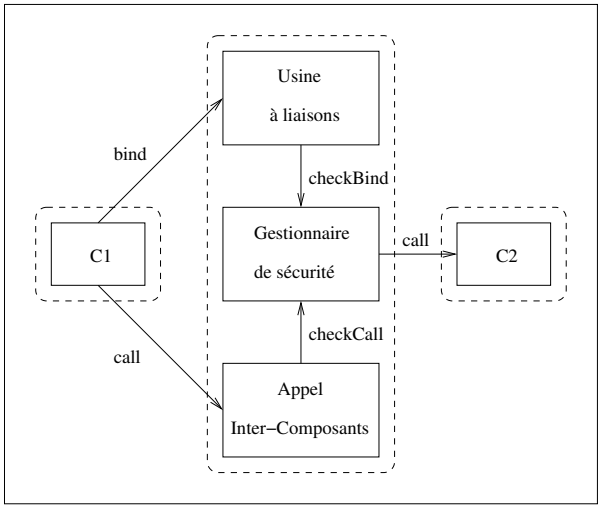

FIG. 6.4 – Architecture d'isolation entre deux composants

est présentée dans la figure 6.5. La matrice d'accès utilisés par l'outil d'isolation est quant à elle détaillée dans la figure 6.6.

```
read(SegC1, SegC1).                read(SegCanevas, SegC1).
write(SegC1, SegC1).               read(SegCanevas, SegC2).
exec(SegC1, SegC1).                read(SegCanevas, SegCanevas).
exec(SegC1, SegCanevas).           write(SegCanevas, SegC1).
                                   write(SegCanevas, SegC2).
                                   write(SegCanevas, SegCanevas).
read(SegC2, SegC2).                exec(SegCanevas, SegC1).
write(SegC2, SegC2).               exec(SegCanevas, SegC2).
exec(SegC2, SegC2).                exec(SegCanevas, SegCanevas).
```

FIG. 6.5 – Spécification de la politique d'accès entre les segments contenant C1 et C2

Pour appeler les méthodes du composant C2, le composant C1 est donc obligé de passer par le canevas sécurisé puisque le segment contenant C2 lui est interdit en exécution. Il utilise donc les méthodes `bind` et `call` qui sont elles situées dans le segment contenant le canevas logiciel auquel le composant C1 a accès en exécution. Cela entraîne les vérifications associées dans le canevas sécurisé via les méthodes `checkBind` et `checkCall`. La politique appliquée par le canevas sécurisé est présentée dans la figure 6.7. Cette politique est simplifiée pour ne présenter que les droits pertinents dans notre exemple.

Une fois que le gestionnaire de sécurité a vérifié que l'appel est bien autorisé, il est exécuté par celui-ci. Cet appel est possible puisque le gestionnaire est dans un segment qui a accès en exécution au segment contenant C2. On voit donc sur un exemple simple comment il est possible d'assurer l'isolation des composants en couplant l'outil d'isolation

	SegC1	SegC2	SegCanevas
SegC1	RWX	∅	X
SegC2	∅	RWX	∅
SegCanevas	RWX	RWX	RWX

FIG. 6.6 – Matrice d'accès pour les segments contenant C1, C2 et le canevas sécurisé

mémoire et le canevas sécurisé.

```
interface(itfC2, "C2", 1, "m2", 0).

component("C1").
bind("C1", itfC2).
call("C1", itfC2, "m2").

component("C2").
export("C2", itfC2, 0).
```

FIG. 6.7 – Spécification de la politique appliquée par le canevas sécurisé

6.3 Modification de l'ordonnancement des accès disque

Afin de vérifier la flexibilité de notre gestionnaire de disque adaptable, nous avons mis en œuvre un scénario d'attaque par déni de service. Un processus désirant lancer une attaque par déni de service contre le gestionnaire de disque émettra vraisemblablement de nombreuses requêtes vers des secteurs choisis pour mettre en défaut l'ordonnanceur de disque. Par exemple, si l'algorithme d'ordonnancement est un ordonnancement par temps de recherche croissant (*Shortest Seek Time First*), il saturera le gestionnaire de requêtes sur des secteurs proches de la position actuelle de la tête de lecture de façon à engendrer une famine pour les autres processus.

La détection d'une telle attaque par le gestionnaire de sécurité est possible grâce à la méthode `getQueue` présentée dans la figure 4.6 et qui permet d'accéder à la file des requêtes. Une telle attaque par déni de service se caractérise par la présence dans cette file d'un grand nombre de requêtes provenant du même processus, ce que le gestionnaire de sécurité peut facilement détecter. Le gestionnaire détermine alors en fonction de la politique de gestion du disque s'il doit réagir. En effet, un grand nombre de requêtes provenant du même processus n'indique pas forcément une attaque puisque le processus en question peut avoir légitimement besoin de beaucoup accéder au disque. Dans le cas d'un algorithme d'ordonnancement par temps de recherche croissant par exemple, en plus de compter le nombre de requêtes émises par chaque processus, le gestionnaire de sécurité prend en compte la proximité des secteurs accédés et leur fréquence d'accès, afin de détecter

un processus lisant en boucle des secteurs proches.

Une fois l'attaque avérée, le gestionnaire de sécurité peut décider de remplacer l'algorithme d'ordonnancement par temps de recherche croissant par un autre assurant l'entrelacement des requêtes en fonction du processus émetteur. Un tel algorithme, que l'on peut qualifier d'équitable, a pour but de garantir que chaque processus sera servi dans un temps fini et d'éviter ainsi une famine. La figure 6.8 illustre de manière comparative la réorganisation d'une file de requêtes pour trois algorithmes d'ordonnancement : par ordre d'arrivée (*First Come, First Served*), par temps de recherche croissant (*Shortest Seek-Time First*) et équitable.

```
FCFS      : [P1R0, P1R3, P1R5, P2R2, P2R3, P2R4, P2R6]
SSTF      : [P1R0, P2R2, P1R3, P2R3, P2R4, P1R5, P2R6]
Équitable : [P1R0, P2R2, P1R3, P2R3, P1R5, P2R4, P2R6]
```

FIG. 6.8 – Réorganisation des requêtes dans la file de l'ordonnanceur de disque

Il est important de noter qu'un tel algorithme n'a pas pour vocation d'accélérer le traitement des requêtes mais de garantir une qualité de service minimum en assurant à chaque processus que ses requêtes seront traitées dans un temps fini comme tous les autres processus. A noter également qu'on parle ici d'accès à des secteurs individuels pour simplifier, mais qu'une implantation efficace telle que nous l'avons réalisée travaillera plutôt avec des blocs de secteurs comme unité d'accès pour minimiser les temps de recherche. En utilisant la méthode `setScheduling`, le gestionnaire de sécurité peut donc remplacer l'algorithme d'ordonnancement en place par cet algorithme équitable.

Une fois l'équité de service rétablie, le gestionnaire de sécurité peut prendre la décision de supprimer les requêtes du processus attaquant en utilisant la méthode `removeRequest`, voir même de notifier à l'ordonnanceur de processus de détruire ce processus. Une telle décision pouvant nécessiter l'analyse de requêtes émises par le processus en question sur un laps de temps important, il est nécessaire de garantir une qualité de service minimum dans le système pendant ce temps, d'où l'intérêt de l'algorithme d'ordonnancement équitable. Dans un système multi-utilisateurs, on peut même imaginer qu'une notification est envoyée à l'administrateur du système, afin de l'informer que tel processus appartenant à tel utilisateur monopolise la bande passante disque et de lui demander la réaction à tenir, ce qui rend d'autant plus important la garantie de la qualité de service minimum pendant le temps nécessaire à la prise de décision par l'administrateur.

Cet exemple montre qu'il est simple d'adapter dynamiquement la politique d'ordonnancement des requêtes disque. Nous avons pris l'exemple d'une attaque par déni de service pour justifier le remplacement de l'algorithme d'ordonnancement mais un tel changement peut être justifié dans d'autres cas. Par exemple, on peut vouloir adapter le système à un type d'application nécessitant un ordonnancement particulier, par exemple une applica-

tion multimédia qui a besoin de garanties sur les temps d'accès au disque pour assurer une qualité de diffusion vidéo convenable. Grâce à la séparation entre la gestion de la politique d'ordonnancement (matérialisée par le code de la fonction de tri passée à la méthode `setScheduling` par le gestionnaire de sécurité) et l'outil servant à la mettre en œuvre (c'est à dire l'ordonnanceur de disque qui fournit les méthodes décrites dans la figure 4.6 et appelle la fonction de tri passée via la méthode `setScheduling`), on assure la flexibilité de l'ordonnancement en rendant simple et transparent le changement d'algorithme. Ce n'est pas le cas dans la plupart des systèmes comme Linux par exemple pour lequel l'algorithme d'ordonnancement fait partie du gestionnaire de disque qui doit donc être reprogrammé si on souhaite changer de politique d'ordonnancement (ce qui d'ailleurs ne permet qu'un changement statique, nécessitant une recompilation du noyau et aucune possibilité de changement dynamique).

6.4 Paramétrage dynamique du gestionnaire réseau

Les outils de sécurisation du gestionnaire réseau sont tous paramétrables dynamiquement pour assurer un maximum de flexibilité. On rappelle ici leurs principales caractéristiques.

Filtrage des paquets

L'outil de filtrage des paquets peut être utilisé pour rejeter les paquets provenant d'adresses IP définies dans une liste. Cette liste est gérée par le gestionnaire de sécurité. Elle est initialement définie à partir de la politique de sécurité spécifiée statiquement comme illustré dans la figure 6.9.

```
ban(194.199.25.28).
ban(192.168.0.0, 16).
```

FIG. 6.9 – Spécification statique de la liste des adresses bannies

Le premier axiome illustre comment il est possible de bannir une adresse IP donnée. Le deuxième définit une plage d'adresses bannies, ici le sous-réseau local 192.168/16. Cette liste peut ensuite être modifiée dynamiquement grâce à l'interface offerte par le gestionnaire de sécurité et présentée dans la figure 6.10.

```
void addBannedAddr(char *IP);
void deleteBannedAddr(char *IP);
```

FIG. 6.10 – Interface de mise à jour dynamique de la liste des adresses bannies

La méthode `addBannedAddr` permet d'ajouter une adresse à la liste des adresses bannies alors que la méthode `deleteBannedAddr` permet d'en supprimer une. Bien évidemment, la politique de sécurité précise quels composants ont le droit d'appeler ces méthodes.

Surveillance de la file de stockage

L'outil de surveillance de la file de stockage est paramétrable simplement en faisant varier le taux de remplissage maximal de la file. Ce taux peut être définit dans la politique de sécurité initiale, comme présenté dans la figure 6.11.

```
maxcapacity(75).
```

FIG. 6.11 – Spécification statique du taux maximal de remplissage de la file

Le taux initial est ici défini à 75% de remplissage. Ce taux peut ensuite être modifié dynamiquement grâce à l'interface présentée dans la figure 6.12.

```
void setMaxCapacity(unsigned int max);
```

FIG. 6.12 – Interface de mise à jour dynamique du taux maximal de remplissage de la file

Délocalisation de la file de stockage

Comme précisé lors de la description de cet outil, la délocalisation de la file de stockage dans l'espace mémoire utilisateur passe par l'utilisation du gestionnaire mémoire s'occupant de cet espace utilisateur plutôt que par l'allocateur mémoire du noyau. L'architecture THINK pouvant être utilisée pour générer n'importe quel type de système, la mise en pratique de cette directive dépend fortement de l'architecture du système en question. Il est par exemple tout à fait possible qu'il n'y ait qu'un unique gestionnaire mémoire dans le système qui gère aussi bien l'espace noyau que l'espace utilisateur. De façon générale, la flexibilité de cet outil revient à permettre au gestionnaire de sécurité d'influer sur la localisation de la file de stockage dans tel ou tel espace mémoire si cela a un sens dans le système en question. Dans l'exemple simple d'un système se consacrant uniquement à l'exécution d'un serveur HTTP, cette flexibilité peut prendre la forme d'un booléen indiquant au gestionnaire mémoire s'il doit allouer la file de stockage dans l'espace mémoire du noyau ou dans celui du processus du serveur HTTP.

Contrôle du débit entrant

L'outil de contrôle du débit entrant peut être paramétré de façon assez similaire à l'outil de surveillance de la file de stockage. Comme détaillé précédemment, la fonction `synRate`

renvoie un booléen indiquant si un paquet doit être rejeté en basant sa décision sur un nombre maximal de paquets à accepter par seconde. Ce nombre maximal de paquets peut être défini statiquement dans la politique de sécurité, comme détaillé dans la figure 6.13.

```
maxrate(10).
```

FIG. 6.13 – Spécification statique du nombre maximal de paquets acceptés par seconde

De façon similaire à l'outil de surveillance de la file de stockage, ce débit maximal peut être modifié dynamiquement grâce à l'interface présentée dans la figure 6.14.

```
void setMaxRate(unsigned int max);
```

FIG. 6.14 – Interface de mise à jour dynamique du nombre maximal de paquets acceptés par seconde

Comme on le voit, tous les outils de sécurisation du réseau proposés peuvent aisément être paramétrés pour correspondre à la politique de sécurité mise en œuvre dans le système. Cela est du encore une fois à la séparation systématique entre la partie fonctionnelle (par exemple le code permettant de rejeter un paquet) de la partie décisionnelle (par exemple la fonction décidant si on doit ou non rejeter un paquet) de l'outil en question.

6.5 Un canevas de gestion de ressources

La gestion des ressources d'un système est un problème complexe sur lequel beaucoup de travaux ont été entrepris. Nous proposons ici un exemple de canevas de gestion de ressources tel que nous l'avons implanté dans l'architecture THINK. Cet exemple a pour but d'illustrer la mise en œuvre de notre philosophie de séparation entre la gestion des politiques et les outils servant à les mettre en œuvre. Nous présentons tout d'abord les caractéristiques requises pour un canevas de gestion des ressources d'un système, avant de détailler l'implantation que nous avons réalisé dans un système THINK et de donner un exemple concret d'utilisation.

6.5.1 Caractéristiques requises

Nous détaillons ci-dessous les caractéristiques requises pour un canevas de gestion de ressources flexible.

6.5.1.1 Architecture de gestion de ressources

L'architecture du canevas de gestion de ressource mis en place est importante car elle influence les performances du système. On donne ci-après les caractéristiques fondamen-

tales de l'architecture à mettre en œuvre.

Coût raisonnable Gérer les ressources du système a pour but d'optimiser l'utilisation de celui-ci par les applications et donc d'obtenir les meilleurs performances possibles. Il serait donc paradoxal de mettre en place un système de gestion de ressources dont le coût serait tel qu'il ralentirait visiblement l'exécution du système. L'architecture et les outils utilisés pour la gestion des ressources doivent donc être les plus discrets possible en terme de coût à l'exécution et d'occupation de ressources (notamment en espace mémoire).

Passage à l'échelle Le système dont on souhaite gérer les ressources n'est pas forcément un système centralisé, il peut s'agir d'un système distribué comme une grappe de stations de travail par exemple. Dans ce cas, le nombre de ressources à traiter peut devenir très important et leur gestion nécessiter des traitements complexes. Il est donc important de mettre en place un système de gestion de ressources capable de passer à l'échelle et de fonctionner aussi bien pour un système centralisé que pour un système distribué composé de nombreux nœuds d'exécution.

6.5.1.2 Gestion des informations

Gérer les ressources d'un système nécessite de savoir collecter et gérer les informations relatives à cette gestion de ressources. Ces informations incluent notamment la connaissance des ressources allouées et disponibles à un instant donné dans le système, ce qui est bien sûr indispensable pour gérer ces ressources. On dégage ci-après quelques points importants concernant cette gestion des informations.

Enregistrement dynamique des ressources Afin de pouvoir gérer les ressources du système, le gestionnaire de ressources doit bien évidemment connaître toutes ces ressources. Dans un système moderne, des ressources peuvent être ajoutées ou bien retirées du système à tout moment (par exemple une imprimante réseau dans un système distribué, un disque dur externe sur une station de travail, voire même pour un système mobile un changement complet de l'environnement lorsqu'il se connecte à un réseau local). Il est donc indispensable de pouvoir enregistrer ou supprimer des ressources à l'exécution du système, sans perturber cette exécution (par exemple sans avoir à redémarrer le système d'exploitation d'une station de travail). Cela impose un gestionnaire de ressource gérant les informations de façon flexible et dynamique.

Format flexible des informations Le format dans lequel sont stockées les informations est important car il doit être suffisamment flexible pour permettre l'enregistrement de

tout type de ressources. De plus, il doit permettre aux clients d'influencer la résolution des requêtes si la politique de sécurité le permet.

Comptage homogène Les attaques par déni de service se basent souvent sur le fait que l'utilisation des ressources n'est pas comptabilisée de la même façon selon que le consommateur en est le noyau du système ou une application. Ainsi, dans certains systèmes, les consommations du noyau ne sont pas sujettes à des quotas comme cela peut être le cas pour les consommations des applications. Dans ce cas, il est en général assez simple pour une application malveillante de provoquer une saturation d'une ressource du système en appelant de façon répétitive les fonctions du noyau qui engendrent une consommation de la ressource en question. Pour pallier cette vulnérabilité, il est nécessaire de mettre en place un système de comptage de l'utilisation des ressources qui ne fasse pas de différence entre le noyau et les applications, de façon à pouvoir appliquer la même politique de quotas à tous les consommateurs du système.

Facturation indirecte Dans la plupart des systèmes, le système de facturation de la consommation des ressources est basé sur la relation entre un consommateur et un producteur. En réalité, une relation de consommation fait souvent intervenir plus de deux protagonistes. Par exemple, une application qui émet une requête SQL vers une base de données consomme de la « ressource base de données ». Pour résoudre cette requête, la base de données aura besoin de consommer du temps processeur, de la mémoire et de la bande passante disque. Dans un système classique, c'est à la base de données que seront facturées totalement ces consommations, alors qu'il peut paraître plus équitable d'en facturer au moins une partie à l'application qui émet la requête. Le même problème se pose d'ailleurs concernant les attaques par déni de service, puisqu'une application malveillante pourrait provoquer la consommation massive de ressources par une autre application sans jamais être elle même inquiétée. Le système de facturation de la consommation de ressources doit dont être capable de construire le graphe des consommations d'une application donnée (ou du noyau) afin de pouvoir la facturer précisément en fonction de la politique de facturation définie par la politique de gestion de ressources.

6.5.1.3 Mise en œuvre de la gestion des ressources

La mise en œuvre de la gestion de ressources dans un système nécessite un algorithme capable de concilier la politique de gestion de ressources définie par le programmeur système et les desiderata des applications clientes exprimées par le biais des requêtes d'allocation envoyées au gestionnaire, avec l'état courant de l'allocation des ressources dans le système. Cette gestion fait intervenir les points détaillés ci-après.

Langage de définition des politiques de gestion de ressources La définition d'une politique de gestion de ressources par le programmeur système est une tâche complexe dans un système évolué. Il est donc intéressant de fournir au programmeur un support pour la définition de politiques de gestion, sous la forme d'un langage lui permettant d'exprimer les contraintes à respecter lors de l'allocation des ressources.

Gestion de politiques globales Dans la plupart des systèmes actuels, la gestion des ressources est réalisée localement, c'est à dire par des gestionnaire indépendants s'occupant chacun d'une ressource donnée. Ce type de gestion est bien sûr insuffisant si l'on veut pouvoir coordonner la gestion de différentes ressources et obtenir une vue globale de l'allocation dans le système. Il est donc nécessaire de mettre en œuvre une collaboration entre les différents gestionnaires locaux, par exemple en les regroupant selon des critères à définir et en les soumettant à la coopération imposée par un gestionnaire de groupe.

Requêtes paramétrables Lorsqu'une application client émet une requête d'allocation, il peut être nécessaire de lui permettre de préciser sa requête de façon à influencer la résolution. Par exemple, dans un système distribué comportant plusieurs imprimantes en réseau pilotées par un gestionnaire d'impression capable de distribuer les requêtes entre les différentes imprimantes selon une politique prédéfinie, un utilisateur peut indiquer au gestionnaire d'impression qu'il souhaiterait que son document soit imprimé sur une imprimante proche de son bureau. Si la politique de gestion des imprimantes impose par exemple que les documents doivent être répartis de façon à équilibrer la charge entre les différentes imprimantes, l'algorithme de répartition pourra alors choisir parmi les imprimantes les moins occupées celle qui se trouve géographiquement la plus proche du bureau de l'utilisateur. Comme le suggère cet exemple, la politique de gestion définie par l'administrateur du système doit bien sûr avoir priorité sur les requêtes des utilisateurs.

6.5.2 Instantiation dans l'architecture Think

Nous détaillons ci-dessous l'instanciation dans l'architecture THINKde ce canevas de gestion de ressources.

6.5.2.1 Architecture de gestion de ressources

Le modèle de gestion de ressources que nous proposons est basé sur un canevas logiciel composé d'une fédération de gestionnaires locaux de ressources. Chaque gestionnaire local gère une ressource donnée en appliquant la politique spécifiée par le programmeur système pour cette ressource. Chaque gestionnaire local peut ainsi être basé sur les algorithmes classiques que l'on trouve dans la littérature et qui sont en général le fruit de nombreuses recherches visant à optimiser leurs performances.

Ces gestionnaires locaux sont regroupés en domaines de ressources autour d'une propriété commune. Chaque groupe est lui même piloté par un gestionnaire de domaine qui coordonne les gestionnaires locaux. Par exemple, dans un système distribué regroupant plusieurs stations de travail et plusieurs imprimantes, chaque imprimante est pilotée par un gestionnaire local, adapté à ses caractéristiques, et tous les gestionnaires locaux sont regroupés dans un domaine de gestion des imprimantes et coordonnés par un gestionnaire de domaine qui peut par exemple faire de l'équilibrage de charge en répartissant les requêtes d'impression des utilisateurs entre les différentes imprimantes, selon la politique fixée par le programmeur système. On peut aussi par exemple regrouper toutes les imprimantes couleurs dans un domaine et les imprimantes noir et blanc dans un autre afin de faciliter la gestion des requêtes entre utilisateurs privilégiés et utilisateurs normaux.

Finalement, les gestionnaires de domaines sont eux-même regroupé dans le domaine englobant tout le système. Ce domaine est contrôlé par un gestionnaire chargé de mettre en œuvre la politique de gestion de ressources globale. Cette architecture est décrite par la figure 6.15 pour l'exemple de la gestion des imprimantes dans un système distribué.

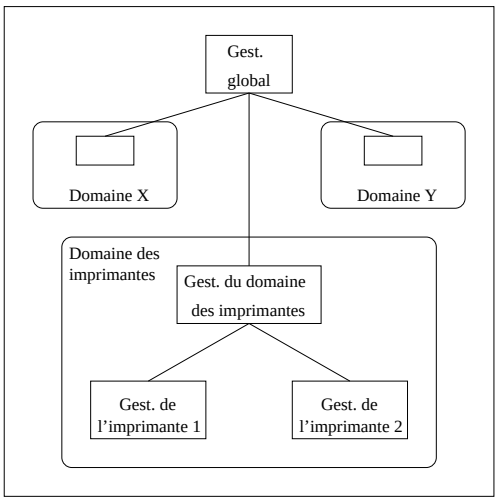

FIG. 6.15 – Exemple de canevas hiérarchique de gestion de ressources

L'intérêt d'une telle architecture hiérarchique est qu'elle est beaucoup plus adaptée à des systèmes de grande taille qu'une architecture centralisée. En effet, dans un système distribué comportant quelques dizaines de stations de travail par exemple, un gestionnaire de ressource centralisé sur une machine gérant les ressources de chaque station du système deviendra vraisemblablement un goulot d'étranglement pour les requêtes émises par les stations de travail. A contrario, en mettant en œuvre notre architecture, chaque station sera gérée localement par un gestionnaire (lui-même gérant une fédération de gestionnaires locaux gérant chacun une ressource de la station), et le gestionnaire global du système

réparti se contentera de mettre en œuvre la coopération entre les différents gestionnaires des stations.

Dans l'architecture THINK, une telle hiérarchisation ne pose pas de problème particulier vu la conception modulaire du système. Un composant client désirant imprimer un document récupérera le nom du gestionnaire de toutes les imprimantes via le courtier et émettra ensuite sa requête d'impression à ce gestionnaire, qui se chargera lui-même de la rediriger vers l'imprimante appropriée.

6.5.2.2 Gestion des informations

Enregistrement dynamique des ressources Afin de permettre la prise en compte de l'ajout et la suppression dynamique de ressources, chaque gestionnaire de domaine intègre un courtier. Ceci permet de s'abstraire de l'implantation des gestionnaires de ressources sous-jacents et rend transparent le remplacement d'un composant non seulement pour les applications mais aussi pour les différents composants du canevas de gestion de ressources. Ainsi, lorsqu'un client émet une requête d'impression au gestionnaire du domaine des imprimantes, celui-ci récupérera le nom de l'imprimante choisie grâce à un nom symbolique du type `lB110a`. Si le gestionnaire de l'imprimante désignée par ce nom symbolique a été changé ou mis à jour, ce changement d'implantation reste transparent pour le gestionnaire de domaine. A noter cependant que si la modification de l'implantation du gestionnaire de l'imprimante change les propriétés de celle-ci, il est nécessaire de le signaler au gestionnaire de domaine afin que celui-ci puisse mettre à jour sa base de connaissance et continuer à rediriger les requêtes de façon optimale.

En plus d'un courtier qui se prête bien à l'enregistrement dynamique de ressources, il est indispensable de pouvoir créer et détruire dynamiquement des liaisons avec les composants du système pour pouvoir utiliser les ressources nouvellement enregistrées ou supprimer les liaisons avec celles qui ont été retirées du système. L'architecture THINK est parfaitement adaptée pour cela grâce à son mécanisme de liaisons flexibles qui permet d'utiliser un composant chargé dynamiquement aussi facilement qu'un composant lié statiquement au noyau.

Format flexible des informations Les informations sur les ressources sont stockées dans les différents gestionnaires sous forme de tuples. L'union des espaces de tuples locaux forme un espace d'information qui fournit une vue de toutes les ressources présentes dans le système et de leur état d'allocation. Chaque tuple est composé d'une référence vers la ressource en question (c'est à dire un nom THINK) et d'un nombre variable de paramètres informatifs. La figure 6.16 donne l'exemple d'un tel tuple.

```
<lB110a_Name, "b&w", "600ppp", "aile B", "PCL6">
```

FIG. 6.16 – Exemple d'un tuple décrivant une imprimante

Comptage homogène Le comptage homogène des ressources est garantit dans un système généré avec l'architecture THINK puisque toutes les entités du système sont des composants. Il n'y a donc pas de différence entre des composants du noyau et ceux des applications du point de vue du gestionnaire de ressources.

Facturation indirecte La facturation indirecte des ressources d'un système nécessite le traçage des consommations et libérations de ressources. L'établissement d'un tel traçage a posteriori est particulièrement complexe puisqu'il implique de retrouver tous les consommateurs indirects d'une ressource en partant de celle-ci, c'est à dire de « remonter » les appels aux méthodes d'allocation et de libération. La méthode de traçage choisie consiste donc à créer des chaînes d'identification au moment de l'allocation ou de la libération d'une ressource. Il suffit pour cela de conserver pour chaque allocation ou libération de ressource un tuple comportant l'identifiant du consommateur, l'identifiant de la ressource et la quantité consommée (négative s'il s'agit d'une libération). Ces informations permettent de construire le graphe des allocations dans le système, qu'il suffit de parcourir en sens inverse pour retrouver tous les consommateurs d'une ressource donnée et ainsi permettre une facturation indirecte de la ressource.

Dans THINK, ceci se traduit par le passage d'un argument supplémentaire à chaque appel à une fonction d'allocation ou de libération d'une ressource. Ce paramètre est un pointeur vers une table stockant les tuples contenant les informations sur les consommations de ressources des composants précédant dans la chaîne de consommation. La figure 6.17 donne un exemple de consommation chaînée.

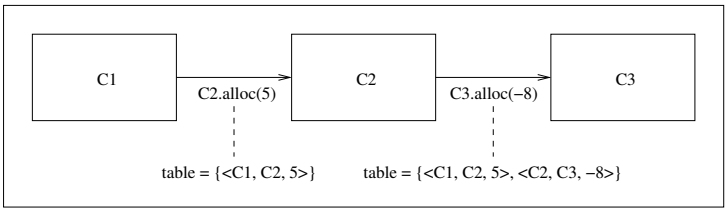

FIG. 6.17 – Exemple de consommation chaînée

Dans cet exemple, le composant C1 consomme 5 unités de la ressource gérée par le composant C2, ce qui est noté dans la table stockant cette chaîne de consommation. La consommation des 5 unités de la ressource gérée par C2 entraîne alors la libération par C2 de 8 unités de la ressource gérée par C3. La chaîne stockée dans la table lors de l'appel

à la méthode de libération de ressources de C3 résume donc l'ensemble des quantités de ressources en jeu dans la consommation chaînée partant de C1.

6.5.2.3 Mise en œuvre de la gestion des ressources

Langage de définition de politiques de gestion de ressources Le langage de définition de la politique de gestion de ressources est aussi important que celui utilisé pour la définition de politiques de sécurité dans le système. Et de la même façon que pour la sécurité, définir un langage permettant d'exprimer n'importe quelle politique de gestion de ressources est un problème très complexe qui dépasse le cadre de ce travail. On utilisera donc des prédicats Prolog pour définir les règles de nos politiques, prédicats qui pourront éventuellement être traduits statiquement en utilisant l'outil servant à la traduction des prédicats liés à la politique de sécurité présenté précédemment, s'ils sont suffisamment simples pour cela bien évidemment. Dans le cas contraire, l'implantation de la politique se fera directement en C et Prolog ne sera utilisé que comme langage de spécification. La figure 6.18 détaille un exemple de règles pour un gestionnaire d'imprimantes connectées à un réseau local.

```
bw(lB110a).
bw(lE110a).
color(cE110b).
aile(lB110a, B).
aile(lE110a, E).
aile(cE110b, E).
user(C1).
root(C2).
print(_c, _p) :- bw(_p).
print(_c, _p) :- root(_c).
```

FIG. 6.18 – Spécification d'une politique de gestion d'imprimantes

Les deux dernières règles précisent les droits d'impression des différents usagers du système. L'avant-dernière donne le droit à n'importe quel composant (par la variable libre `_c`) d'imprimer sur une imprimante noir et blanc. La dernière donne le droit aux composants privilégiés (c'est à dire appartenant à l'utilisateur `root`) d'imprimer sur n'importe quelle imprimante (par la variable libre `_p`). Cela a donc pour effet d'interdire aux utilisateurs non-privilégiés d'imprimer sur les imprimantes couleurs. A noter que ces deux règles ne sont pas compilables par notre outil puisqu'il ne s'agit pas d'axiomes et qu'elles devront donc être codées par des fonctions appropriées dans le gestionnaire d'impression.

Gestion de politiques globales L'architecture hiérarchique présentée ci-dessus permet la définition de politique de gestion de ressources globales. En effet, les gestionnaires de do-

maines (et a fortiori le gestionnaire global du système) contrôlent les gestionnaires locaux s'occupant chacun d'une ressource donnée. Cela impose bien évidemment une communication entre les différents niveaux de gestion. Par exemple, un utilisateur d'un système réparti comportant plusieurs imprimantes émet une requête d'impression d'un document vers le gestionnaire du domaine regroupant toutes les imprimantes. En fonction de la politique de gestion globale des imprimantes, la requête est ensuite redirigée vers le gestionnaire de l'imprimante sélectionnée pour imprimer le document.

Requêtes paramétrables Le paramétrage des requêtes est réalisé simplement en passant en argument de chaque requête un tuple contenant des contraintes à respecter par le resolveur de requêtes du gestionnaire contacté. La politique de gestion précise dans quelle mesure les desiderata des clients doivent être pris en compte et avec quelle priorité par rapport aux contraintes imposées par la politique de gestion elle-même. Par exemple, dans un système réparti auquel sont connectées plusieurs imprimantes, le client qui émet une requête d'impression peut préciser l'imprimante sur laquelle il souhaite faire imprimer son document. La politique précise quelles sont les imprimantes autorisées pour telle ou telle catégorie de clients, et comment les requêtes doivent être réparties entre les différentes imprimantes. Ainsi, si la politique de gestion impose une politique d'équilibrage de charge entre les différentes imprimantes et que l'imprimante sélectionnée par le client est occupée, le gestionnaire imprimera le document sur une autre imprimante et le notifiera au client.

6.5.3 Exemple d'utilisation

On détaille ici un exemple de gestion de ressources dans un système THINK. L'exemple choisi concerne la gestion d'imprimantes connectées à un réseau local. Un composant client (par exemple un éditeur de texte) émet un requête d'impression vers le composant gestionnaire du domaine des imprimantes qui, en fonction des paramètres fournit par le client et de la politique de gestion de ressources, redirige la requête vers le gestionnaire approprié. Cette architecture hiérarchique est détaillée dans la figure 6.19.

Dans cet exemple, le découpage en domaines a été effectué selon la capacité des imprimantes à imprimer en couleur. On a donc deux sous-domaines dans le domaine d'impression, le sous-domaine des imprimantes noir et blanc et celui des imprimantes couleurs. Chaque domaine est géré par un gestionnaire qui est capable de rediriger les requêtes lui parvenant vers les imprimantes qu'il gère. Le système comprend ici 3 imprimantes, deux noir et blanc (`lB110a` et `lE110a`) et une couleur (`cE110b`).

La figure 6.20 présente l'interface du gestionnaire de domaine d'impression.

La méthode `print` permet à un client d'émettre une requête d'impression d'un fichier désigné par son nom (`fileName`). Le client peut préciser sa requête en passant un tuple d'arguments (`args`) en paramètre. La méthode `register` permet à un sous-gestionnaire

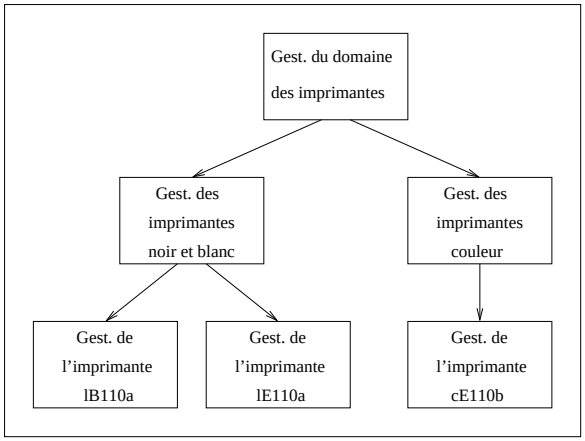

FIG. 6.19 – Exemple de canevas de gestion des imprimantes

```
void print(char *fileName, Tuple args);
void register(char *symbName, Name resource, Tuple properties);
void unregister(char *symbName);
void update(char *symbName, Name resource, Tuple properties);
```

FIG. 6.20 – Interface du gestionnaire du domaine d'impression

d'impression de s'enregistrer auprès du gestionnaire du domaine des imprimantes. Le sous-gestionnaire fournit lors de l'appel à cette méthode un nom symbolique (`symbName`) qui est utilisé par le gestionnaire pour désigner de façon simplifiée le sous-gestionnaire (ce nom peut être assimilé à un nom statique si l'on souhaite s'en servir lors de la définition de la politique de gestion par exemple). Le nom THINK (`resource`) du sous-gestionnaire est également fourni pour permettre au gestionnaire de rediriger la requête vers le sous-gestionnaire. Enfin, les caractéristiques du sous-gestionnaire (`properties`) sont communiquées au gestionnaire du domaine afin qu'il puisse rediriger les requêtes vers le sous-gestionnaire possédant les caractéristiques requises. La méthode `unregister` permet de notifier au gestionnaire du domaine que le sous-gestionnaire désigné par son nom symbolique (`symbName`) n'est pas accessible et qu'il ne doit plus être pris en compte lors de la résolution des requêtes. Enfin, la méthode `update` permet de modifier les caractéristiques d'un sous-gestionnaire déjà enregistré, par exemple si on a mis à jour le sous-gestionnaire en question pour ajouter des fonctionnalités.

L'interface des sous-gestionnaires est en fait identique à celle du gestionnaire présenté dans la figure 6.20. Les méthodes ont le même rôle, sauf qu'il ne s'agit plus de sous-gestionnaires qui s'enregistrent ou se désenregistrent, mais de gestionnaires d'imprimantes.

La figure 6.21 présente l'interface d'un gestionnaire d'imprimante.

La méthode `print` a le même rôle que celle du gestionnaire du domaine d'impression

```
void print(char *fileName, Tuple args);
```

FIG. 6.21 – Interface d'un gestionnaire d'imprimante

et des sous-gestionnaires, à la différence qu'elle ne redirige pas la requête mais l'exécute.

La résolution des requêtes se fait elle aussi hiérarchiquement. Par exemple, le composant client émet la requête `print("fichier.ps", <couleur, aile B>).` pour demander l'impression du fichier `fichier.ps` sur une imprimante couleur de l'aile B de son lieu de travail. Le gestionnaire du domaine d'impression résout la requête en prenant en compte les paramètres fournis par le client et la politique de gestion des impressions. Cette politique a bien entendu priorité sur les desiderata du client. Par exemple, si la politique précise que seuls les client privilégiés ont le droit d'imprimer en couleur, la requête sera redirigée vers le gestionnaire des imprimantes noir et blanc malgré le paramètre `couleur` de la requête. Une fois la requête reçue par le gestionnaire du sous-domaine des imprimantes noir et blanc, celui-ci analyse a son tour la requête et note que le client souhaite imprimer sur une imprimante de l'aile B de son lieu de travail. En supposant que la politique de gestion locale l'autorise, la requête est alors redirigée vers l'imprimante `lB110a`.

6.5.4 Évaluation

Cet exemple de canevas de gestion de ressources illustre la mise en œuvre de la philosophie présentée tout au long de ce travail. En effet, en séparant la gestion de la politique de gestion de ressources des outils permettant de la mettre en œuvre, on garantit la flexibilité de la gestion. Ainsi, dans le canevas présenté ci-dessus, les gestionnaires de groupes ont la charge de piloter les gestionnaires de ressources en se basant sur la politique définie. Il est donc aisé de remplacer un gestionnaire de ressource sans avoir à modifier le gestionnaire de groupe appliquant la politique concernant cette ressource, et à l'inverse, de modifier la politique de gestion du domaine de ressource sans toucher aux gestionnaires locaux. Concernant la politique mise en œuvre par ces gestionnaires locaux, même si cette politique est codée dans le gestionnaire comme c'est souvent le cas dans la plupart des systèmes, elle peut être modifiée simplement en remplaçant le gestionnaire, ce qui est facilité par l'architecture modulaire de THINK.

De plus, ce canevas de gestion de ressource utilise le canevas sécurisé implanté. Par exemple, l'outil de facturation indirecte des consommations de ressources nécessite d'être sécurisé pour pouvoir remplir son rôle efficacement. En effet, afin de s'assurer qu'un composant utilisant une ressource ne peut pas modifier la chaîne de consommation en récupérant le pointeur vers la table de stockage, cette table est allouée dans un espace mémoire accessible uniquement par le gestionnaire de sécurité. Cet espace peut être par exemple être isolé grâce à notre mécanisme d'isolation logicielle. C'est donc la méthode `call` du canevas

qui manipule le pointeur en question et mets à jour la table. Sans une telle sécurisation, n'importe quel composant pourrait aisément modifier la table pour par exemple supprimer sa référence de la chaîne et cacher sa consommation de ressources au système.

La définition de la politique de gestion des ressources du système peut être basée sur le même langage que la politique de sécurité. Dans notre exemple, nous utilisons Prolog pour spécifier une politique qui est ensuite implantée en C, ce qui peut aussi bien s'appliquer à la définition d'une liste de droits d'accès qu'à un ensemble de contraintes sur la répartition des ressources d'un système. Bien que le présent travail ne porte pas sur la spécification d'un langage de définition de politiques génériques on peut utiliser nos résultats partiels et notamment l'outil de génération de code C à partir d'une liste d'axiomes Prolog dans les deux cas.

6.6 Sécurisation de réseaux actifs

Les réseaux actifs sont un domaine dans lequel la protection est particulièrement importante. En plus d'assurer le routage des paquets, les nœuds composant un réseau actif offrent une puissance de calcul pouvant être utilisée à mauvais escient par des utilisateurs malveillants. On présente donc ici une architecture de réseau actif que nous avons réalisée à l'aide de l'architecture THINK et montrons comment elle peut être sécurisée en utilisant le canevas et les outils de protection implantés dans THINK. Cette architecture a été spécifiée en collaboration avec Aline Senart dans le cadre de son travail sur la reconfiguration dynamique de systèmes THINK [Sen03].

6.6.1 Présentation des réseaux actifs

Les réseaux actifs sont des réseaux dont les nœuds peuvent exécuter du code arbitraire. En effet, à la différence des paquets transitant sur les réseaux classiques qui ne contiennent que des données, les paquets « actifs », souvent appelés capsules, contiennent également du code destiné à être exécuté sur les nœuds que le paquet traverse. Ce code est le plus souvent destiné à la reconfiguration dynamique du routeur, bien qu'en théorie n'importe quel programme puisse être exécuté, transformant ainsi le réseau en une véritable machine de calcul parallèle.

6.6.1.1 Principes

L'architecture d'un nœud actif est en général divisée en plusieurs couches. La couche applicative comprend les différents programmes qui s'exécutent sur le nœud. Chaque programme s'exécute dans un environnement d'exécution qui lui fournit une interface d'accès aux ressources et aux services du système. Plusieurs programmes peuvent s'exécuter dans le même environnement d'exécution ce qui permet une collaboration inter-applications.

Plusieurs environnements d'exécution peuvent cohabiter sur le même nœud, afin de fournir des services différents aux applications. Par exemple, un nœud peut comprendre un environnement d'exécution « natif » pour permettre l'exécution de binaires et un environnement d'exécution Java qui s'apparentera à une machine virtuelle Java pour l'exécution d'Applets. Les différents environnements d'exécution fonctionnent au dessus du système d'exploitation du nœud qui a en charge l'isolation des environnement et la répartition des ressources entre ceux-ci. Ces concepts sont illustrés dans la figure 6.22.

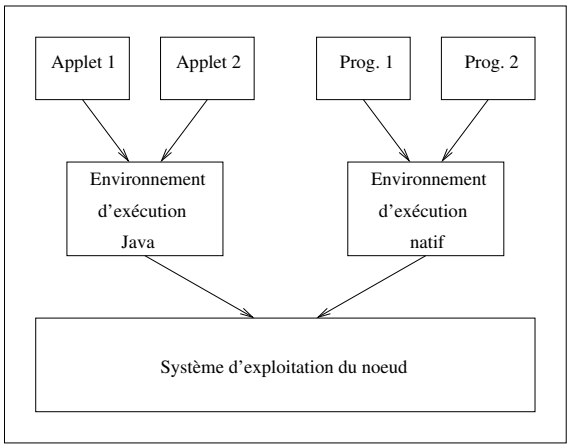

FIG. 6.22 – Exemple d'architecture d'un nœud actif

Lorsqu'un paquet arrive sur un nœud, son en-tête est analysé afin de déterminer vers quel environnement d'exécution il doit être redirigé. Pour préserver la compatibilité avec les réseaux classiques, les paquets passifs sont redirigés vers un environnement d'exécution contenant une application de routage similaire à celle trouvée dans un routeur classique.

6.6.1.2 L'interface NodeOS

Pour assurer la portabilité des environnements d'exécution sur les différents systèmes d'exploitation utilisés dans les routeurs, une interface a été spécifiée par le groupe de travail sur les réseaux actifs [Ca98]. Cette interface, connue sous le nom d'interface NodeOS, est désormais devenue standard dans le domaine des réseaux actifs. Elle définit quatre abstractions utilisables par les environnements d'exécution : les réserves de processus légers (de l'anglais *thread pools*), les réserves de mémoire (de l'anglais *memory pool*), les canaux (de l'anglais *channel*) et les domaines (parfois désignés par le terme flux, de l'anglais *flow*, bien que ce terme puisse être confondu dans un contexte réseau avec le flux de paquets traversant le routeur).

Les réserves de processus et de mémoire permettent au système d'exploitation d'allouer les ressources processeur et mémoire de la machine aux différents environnements

d'exécution. Ces abstractions sont particulièrement utiles pour le comptage des ressources, un mécanisme nécessaire à la mise en œuvre de politiques de gestion de ressources. Les canaux représentent les flux de paquets destinés à un environnement d'exécution déterminé grâce à l'en-tête des paquets. Il existe trois types de canaux, les canaux d'entrée (`inChan`) par lesquels les paquets transitent de la couche de communication du système d'exploitation vers l'environnement d'exécution, les canaux de sortie (`outChan`) qui assurent le transport des paquets depuis l'environnement d'exécution vers le système d'exploitation qui les émettra sur le réseau, et les canaux de court-circuit (`cutChan`) qui transportent les paquets du point d'entrée de la couche de communication du système d'exploitation vers le point de sortie, sans passer par un environnement d'exécution.

Un domaine est une abstraction qui englobe des réserves de processus, des réserves de mémoire et des canaux. La figure 6.23 présente un exemple typique de domaine, représenté par la zone entourée en pointillés. Un domaine peut contenir un ou plusieurs sous-domaines qui forment une hiérarchie pouvant notamment être utilisée pour mettre en place une gestion hiérarchique des ressources dans un environnement d'exécution.

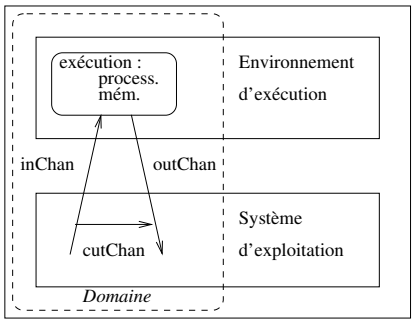

FIG. 6.23 – Exemple de domaine dans un nœud actif

6.6.1.3 La sécurité dans les réseaux actifs

La sécurité est un problème encore plus crucial dans un réseau actif que dans un réseau classique. En effet, permettre l'exécution de code arbitraire sur un nœud rend celui-ci vulnérable à des programmes malveillants ou simplement bogués, de façon assez similaire à ce que risque un système extensible lorsqu'on y charge un module d'origine douteuse. La problématique de la sécurité s'articule autour des différents aspects présentés ci-dessous.

Authentification Avant de pouvoir définir les permissions accordées aux différents programmes pouvant traverser le nœud, il est bien entendu nécessaire de pouvoir les identifier. Ce problème est beaucoup plus complexe dans un réseau que dans un système centralisé où un objet peut être identifié depuis sa création jusqu'à sa destruction. Dans un réseau, des

machines émettrices de paquets actifs peuvent être ajoutées et retirées à tout moment sans que les routeurs n'en soient forcément informés. De plus, un paquet peut être modifié par les routeurs qu'il a traversés et qui pourraient avoir modifié le code initial pour le rendre dangereux. Pour résoudre ce problème, la plupart des systèmes emploient des techniques de signature digitale, comme les signatures PGP [Zim95] par exemple, qui permettent à un nœud de s'assurer que le paquet provient bien d'un émetteur connu et dont il possède la clé publique. Cela permet aussi de s'assurer que le paquet n'a pas été modifié depuis son émission.

Protection du routeur Un code malveillant pourrait compromettre le bon fonctionnement du routeur en modifiant des données ou en consommant toutes les ressources, de façon similaire aux risques encourus par un système classique. Les mêmes techniques que celles présentées pour les systèmes d'exploitation peuvent donc être mises en œuvre. Beaucoup de systèmes fonctionnent avec des programmes écrits dans un langage destiné à être interprété et permettant des vérifications à l'exécution comme Java par exemple. D'autres utilisent des techniques d'isolation matérielle ou logicielle pour encapsuler le code exécuté et lui interdire d'accéder à des zones mémoire critiques. De façon similaire, on retrouve les mêmes techniques de protection de la qualité de service que dans les systèmes centralisés.

Protection du réseau Un problème de sécurité spécifique aux réseaux concerne la protection de la qualité de service sur le réseau. Un exemple simple d'une telle attaque contre le réseau consiste à transmettre à un routeur un paquet actif contenant un programme dont le seul but est d'émettre un paquet actif contenant le même programme vers tous les nœuds voisins. Vu l'interconnexion des routeurs dans un réseau, cela provoquerait en quelques secondes une inondation de paquets qui dégraderait considérablement les performances du réseau. De telles attaques sont en général contrées par la mise en œuvre dans le routeur de politiques de gestion du réseau qui interdisent par exemple à un programme d'émettre plus d'un paquet actif en sortie du routeur.

6.6.2 Sécurisation d'un nœud de réseau actif

6.6.2.1 Architecture

Nous détaillons ci-dessous l'architecture d'un environnement d'exécution sécurisé réalisé grâce au canevas sécurisé de l'architecture THINK et aux outils que nous avons présentés précédemment. L'environnement d'exécution sécurisé que nous présentons est basé sur un noyau de système d'exploitation reconfigurable réalisé avec THINK et détaillé dans [Sen03]. L'environnement d'exécution est lui aussi réalisé à l'aide des composants THINK présentés dans la figure 6.24.

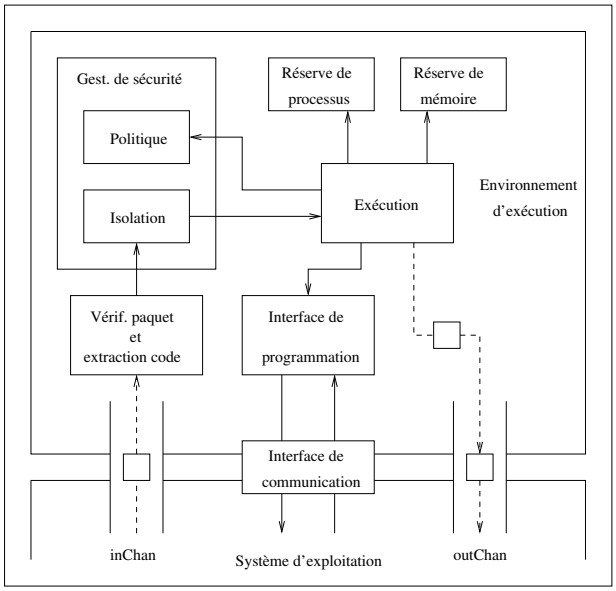

FIG. 6.24 – Environnement d'exécution sécurisé

Un paquet actif arrivant dans l'environnement d'exécution via le canal `inChan` passe tout d'abord par un composant chargé de vérifier que le format du paquet est correct et si c'est le cas, d'en extraire le code à exécuter et les données qui peuvent y être attachées. Dans notre cas, l'environnement supporte l'exécution de programmes écrits en binaire pouvant s'exécuter sur la machine sous-jacente, ce qui suppose que tous les routeurs du réseau soient des machines homogènes. Cette supposition n'est pas réaliste pour des paquets actifs devant circuler sur Internet mais peut être mise en œuvre sans problème pour un réseau local. Les paquets destinés à être émis sur Internet contiennent généralement du code dans un langage destiné à être interprété comme Java ou PLAN [HK99] pour pallier l'hétérogénéité des routeurs.

Une fois le code extrait, celui-ci passe par notre outil d'isolation mémoire logicielle. La politique d'isolation mise en œuvre pour garantir une exécution sûre du code extrait du paquet consiste à interdire tous les accès mémoire en dehors de la zone dans lequel il sera exécuté, à l'exception des appels vers la zone mémoire contenant l'interface de programmation (de l'anglais *Application Program Interface*) de l'environnement d'exécution. Le programme pourra donc communiquer avec l'environnement d'exécution via cette interface mais ne pourra pas accéder à des données ou du code appartenant aux autres programmes s'exécutant dans le même environnement.

Une fois le code isolé, il est exécuté dans une zone mémoire réservée, dans laquelle sont alloués tous les processus et les espaces mémoire dynamiques dont il peut avoir besoin au cours de son exécution. Ces processus et espaces mémoire sont pris dans les réserves de

processus et de mémoire de l'environnement d'exécution. La politique de sécurité influence l'allocation des processus et de la mémoire aux différents programmes s'exécutant dans l'environnement d'exécution afin de permettre la mise en place de politique de gestion de ressources. De plus, les accès aux méthodes de l'interface de programmation de l'environnement d'exécution se font via le canevas sécurisé de THINK (puisque cette interface est une interface THINK) ce qui permet d'effectuer des contrôles sur les méthodes appelées via la méthode `checkCall` du gestionnaire de sécurité.

Des contrôles plus fins peuvent être effectués via la méthode `checkBind` si on divise l'interface de programmation en plusieurs interfaces THINK, ce qui permet d'interdire à certains programmes de se lier avec certaines interfaces. Par exemple, on peut diviser l'interface de programmation de l'environnement en une interface fournissant des méthodes de mise à jour de données stockées dans le routeur (par exemple des informations sur l'état des différentes branches du réseau), et une deuxième comprenant des méthodes permettant de paramétrer le routeur lui-même, c'est à dire mettre à jour le programme de routage voire même reconfigurer le système d'exploitation et les environnements d'exécution dans le cas d'un noyau de système reconfigurable. Les méthodes de la deuxième interface sont bien évidemment beaucoup plus sensibles que celles de la première, et on peut vouloir en restreindre l'accès à des paquets provenant par exemple de l'administrateur du réseau (authentifié grâce aux techniques de signature électronique présentées plus haut). Enfin, si on permet à un programme extrait d'un paquet actif de modifier l'interface de programmation de l'environnement d'exécution, pour ajouter de nouvelles fonctionnalités ou corriger des erreurs par exemple, il est nécessaire d'utiliser la méthode `checkExport` du gestionnaire de sécurité pour vérifier si la mise à jour en question est bien autorisée par la politique de sécurité.

L'environnement d'exécution dialogue avec le système d'exploitation sous-jacent via une interface de communication. Cette interface permet notamment aux méthodes de l'interface de programmation de l'environnement d'exécution de fournir des points d'accès vers le système d'exploitation pour les programmes extraits des paquets actifs. Ainsi un utilisateur autorisé, tel que l'administrateur du réseau par exemple, peut envoyer un programme provoquant non seulement une modification de l'environnement d'exécution, mais également du système d'exploitation. Cela est notamment utile pour permettre la reconfiguration dynamique du système d'exploitation, comme présenté ci-dessous.

Si l'exécution du programme provoque l'émission de nouveaux paquets sur le réseau, ceux-ci sont transmis au système d'exploitation sous-jacent via le canal `outChan`. Pour éviter les attaques contre la qualité de service du réseau (par émission massive de paquets), il est nécessaire que la méthode de l'interface de programmation de l'environnement d'exécution servant à l'émission de paquets communique avec le gestionnaire de sécurité. La politique de sécurité permet en effet d'exprimer de façon simple les restrictions sur le

nombre de paquets pouvant être émis selon les programmes. Par exemple, un programme identifié comme provenant de l'administrateur du réseau pourra être autorisé à émettre autant de paquets qu'il le désire, alors qu'un programme provenant d'une autre source ne pourra émettre qu'un paquet, voire même aucun.

6.6.2.2 Exemple : reconfiguration sécurisée d'un système

Une application de notre architecture sécurisée pour les réseaux actifs consiste à offrir à un administrateur réseau la possibilité de reconfigurer dynamiquement les routeurs actifs de son réseau local. Cela peut être utile par exemple pour mettre à jour le programme de routage, voir même l'environnement d'exécution ou le système d'exploitation. Cela permet d'effectuer des mises à jour automatisées et à distance, ce qui facilite grandement la tâche de l'administrateur notamment sur un réseau comprenant un nombre important de nœuds potentiellement éloignés géographiquement.

Nous présentons ici un exemple simple de reconfiguration : l'administrateur d'un réseau local désire mettre à jour le gestionnaire réseau de tous les routeurs de son réseau pour remplacer le gestionnaire existant par un gestionnaire intégrant nos outils de protection contre les attaques par inondation. Pour cela, il diffuse un paquet actif contenant le code de reconfiguration et le nouveau gestionnaire réseau sécurisé, code qui va utiliser les méthodes de l'interface de programmation de l'environnement d'exécution pour effectuer la reconfiguration. Nous détaillons ci-dessous l'implantation des différents composants détaillés dans la figure 6.24 et intervenant dans cette reconfiguration.

Les paquets actifs Nous avons défini un format simple pour les paquets actifs traités par notre environnement d'exécution. Ce format est détaillé dans la figure 6.25.

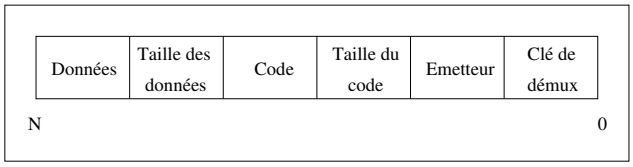

FIG. 6.25 – Format des paquets actifs

La clé de démultiplexage sert au système d'exploitation pour diriger le paquet actif vers l'environnement d'exécution adapté. Une telle clé peut par exemple préciser le langage dans lequel est écrit le code inclus dans le paquet. Elle n'a théoriquement plus d'utilité une fois que le paquet est arrivé dans l'environnement d'exécution, bien qu'il soit raisonnable de la vérifier avant d'accéder aux autres champs du paquet pour détecter une éventuelle erreur d'aiguillage synonyme d'une erreur de programmation du système d'exploitation.

Le champ `émetteur` permet d'identifier l'émetteur du paquet. Dans le cas général, cette authentification peut s'avérer complexe et nécessiter l'utilisation de signatures électroniques. Dans le cadre de notre exemple, nous considérons que le réseau sur lequel transitent les paquets est un réseau local sûr dont tous les routeurs sont administrés par l'émetteur des paquets actifs. On se contente donc d'une clé sur 128 bits ce qui rend statistiquement très peu probable la possibilité pour un utilisateur malveillant de fabriquer lui-même cet identifiant.

La taille du code est précisée dans le paquet car ce code peut être de taille variable (le paquet actif lui-même est de taille fixe pour simplifier sa gestion, ce qui impose de gérer la fragmentation du code et des données entre plusieurs paquets si l'on souhaite transmettre des programmes de taille importante). Le code lui-même est transmis en binaire dans notre exemple, bien que le langage choisi n'ait pas d'influence sur le format du paquet. La figure 6.26 détaille le code binaire envoyé par l'administrateur dans le paquet actif pour lancer la reconfiguration. Ce code est présenté ici en assembleur afin d'en simplifier la lecture.

```
mflr 17                                 //

lis 18,hi(point d'entree de l'API)      //
ori 18,18,lo(point d'entree de l'API)   //
mtlr 18                                 // r18 = replaceComp();
li 18,idf(replaceComp)                  //
blrl                                    //

cmpwi 0,18,1                            // if (r18 == 1)
beq 0,exit                              //   goto exit;

lis 18,hi(point d'entree de l'API)      //
ori 18,18,lo(point d'entree de l'API)   //
mtlr 18                                 // forwardPacket();
li 18,idf(forwardPacket)                //
blrl                                    //

exit:                                   //
mtlr 17                                 // return;
blr                                     //
```

FIG. 6.26 – Code de lancement de la reconfiguration

Ce code a pour effet d'appeler tout d'abord la méthode `replaceComp` de l'interface de programmation. Cette méthode a pour but de demander le remplacement d'un composant par un autre. Elle renvoie 0 en cas de succès et 1 en cas d'échec. Le code appelle ensuite la méthode `forwardPacket` qui a pour effet de transmettre le même paquet actif à tous les routeurs adjacents au routeur courant. Cela permet à l'administrateur du réseau de

n'envoyer qu'un seul paquet actif qui se propagera à tous les routeurs. Une des raisons pour lesquelles la méthode `replaceComp` renvoie 1 étant que le remplacement demandé a déjà été effectué, on s'assure de la terminaison de la mise à jour du réseau en ne transmettant le paquet aux routeurs adjacents que si l'appel à la méthode `replaceComp` réussit en renvoyant 0.

On notera que l'appel aux méthodes `replaceComp` et `forwardPacket` ne se fait pas directement dans le code binaire présenté dans la figure 6.26. On passe par un point d'entrée bien connu de l'interface de programmation qui redirige les appels aux méthodes voulues en fonction des paramètres passés dans le registre 18. Cela évite à l'administrateur du réseau d'avoir à connaître les adresses de toutes les méthodes de l'interface lorsqu'il programme le code du paquet actif. Ce code effectue donc un simple branchement vers le point d'entrée de l'interface de programmation, qui utilise ensuite le canevas sécurisé de THINK pour se lier à l'interface de programmation et appeler la méthode choisie.

Le paquet peut également contenir des données utiles lors de l'exécution du code. Dans notre cas, les données du paquets actifs sont le nom symbolique du composant à remplacer, le nom symbolique du nouveau composant, et le code binaire de ce nouveau composant qu'on va devoir charger dans le système. Le format des données dépend complètement de l'exécution du code extrait du paquet actif et il appartient à l'émetteur du paquet de s'assurer que ce format est tel que les méthodes de l'interface de programmation l'attendent.

L'interface de programmation C'est cette interface que le code extrait du paquet actif va utiliser pour communiquer avec l'environnement d'exécution et, via l'interface de communication, avec le système d'exploitation. La figure 6.27 présente les méthodes de cette interface intervenant dans la reconfiguration du système.

La méthode `replaceComp` permet au code extrait du paquet actif de demander le remplacement d'un composant désigné par son nom symbolique par un nouveau composant dont le nom symbolique est lui aussi passé en argument. Le paramètre `secret` permet d'identifier l'émetteur du paquet et provient de l'identificateur de l'émetteur extrait du paquet actif.

L'authentification de l'émetteur est bien entendu très importante pour permettre la mise en œuvre de la politique de sécurité dans l'environnement d'exécution. Ainsi, avant d'effectuer le remplacement du composant choisi, la méthode `replaceComp` doit vérifier que ce remplacement est bien autorisé par la politique de sécurité. Pour cela, elle utilise la méthode `checkReplaceComp` exportée par le gestionnaire de sécurité et dont le prototype est similaire à celui de `replaceComp`. Cette méthode vérifie simplement que l'émetteur identifié a bien le droit de remplacer le composant désigné par son nom statique en vérifiant la présence de la capacité appropriée dans la politique de sécurité spécifiée. Une telle autorisation peut être simplement exprimée par l'axiome Prolog : `unload(admin, "netmgr").`

```
int replaceComp(Secret secret,
                char *oldCompSymbName,
                char *newCompSymbName,
                char *binComp) {
  if (checkReplaceComp(secret, oldCompSymbName, newCompSymbName) == 1) {
    return 1;
  }
  if (unloadComp(oldCompSymbName) == 1) {
    return 1;
  }
  if (loadComp(newCompSymbName, binComp) == 1) {
    return 1;
  }
  return 0;
}

void forwardPacket(Secret secret) {
  if (checkForwardPacket(secret) == 1) {
    return;
  }
  forwardToAll();
}
```

FIG. 6.27 – Interface de programmation

qui permet à l'administrateur du réseau de remplacer (c'est à dire décharger) le gestionnaire de réseau `netmgr`.

Si ce remplacement est autorisé, la politique de sécurité est modifiée de façon à interdire le remplacement de l'ancien composant. En effet, cela n'aurait plus de sens de laisser l'autorisation de remplacer un composant qui n'est plus dans le système. Un effet de bord intéressant de cette modification de la politique est qu'en cas de deuxième réception du même paquet actif, la demande de remplacement sera refusée et le paquet n'aura aucun effet. Comme il ne sera pas non plus retransmis aux autres routeurs, cela permet de terminer la mise en jour des routeurs du réseau.

En supposant que la politique autorise bien le remplacement de l'ancien composant, on peut également vouloir vérifier qu'on a bien le droit de le remplacer par le nouveau composant donné. Mais cela implique de connaître à l'avance tous les composants pouvant éventuellement remplacer ceux déjà présents dans le système. C'est à la politique de sécurité de préciser cela et le cas échéant de maintenir la liste de tous les composants autorisés à en remplacer d'autres. Dans notre exemple, on autorise l'administrateur du réseau à remplacer n'importe quel composant présent dans le système par n'importe quel autre et on suppose qu'il s'assurera lui-même de la cohérence de ce remplacement, ce qui s'exprime en Prolog par l'axiome : `load(admin, _x).`.

Pour effectuer le remplacement, la méthode `replaceComp` se base sur les méthodes

offertes par le système d'exploitation reconfigurable, en passant par l'interface de communication. La méthode `unloadComp` de l'interface de communication permet d'appeler les mécanismes du système permettant de décharger le composant dont le nom symbolique est donné en argument. De façon similaire, la méthode `loadComp` appelle les mécanismes permettant de charger dans le système un composant dont le nom symbolique et la représentation binaire sont passés en paramètres. Les mécanismes système mis en jeu sont décris en détails dans [Sen03].

La méthode `replaceComp` renvoie 0 si le remplacement du composant réussit et 1 autrement. Comme décrit ci-dessus, la méthode `replaceComp` peut renvoyer 1 pour plusieurs raisons : si la politique de sécurité interdit le remplacement (par exemple parce qu'il a déjà été effectué), si le déchargement de l'ancien composant a échoué, ou si le chargement du nouveau composant n'a pas abouti. Dans tous les cas, cela aura pour effet de ne pas transmettre le paquet actif aux routeurs adjacents. Une gestion des erreurs plus sophistiquée pourrait émettre un paquet de notification à l'administrateur du réseau, notamment en cas d'erreur provenant des mécanismes système de reconfiguration car l'échec d'un de ces mécanismes pourrait laisser le système dans un état instable.

La méthode `forwardPacket` a pour effet de provoquer la ré-émission du paquet actif via le canal `outChan`, à destination de tous les routeurs adjacents au routeur sur lequel s'exécute le code. Avant de permettre cette émission, la politique de sécurité est consultée par le biais de la méthode `checkForwardPacket`. Cette vérification est nécessaire pour éviter qu'un utilisateur malveillant puisse saturer le réseau en émettant un paquet actif contenant un code provoquant une multiplication des émissions de paquets. Dans notre exemple, on autorise seulement l'administrateur du réseau à utiliser la méthode `forwardPacket` et on refuse toutes les demandes provenant d'autres utilisateurs, ce qu'on exprime simplement par l'axiome Prolog : `emit(admin, _x).` où la variable libre `_x` représente le nombre maximum de paquets pouvant être émis. Si l'autorisation est accordée par la politique de sécurité, la méthode `forwardToAll` de l'environnement d'exécution est appelée afin de transférer le paquet actif à tous les routeurs adjacents via le canal `outChan`.

L'interface de communication Cette interface permet la communication entre l'environnement d'exécution et le système d'exploitation. La figure 6.28 présente les méthodes de cette interface utiles pour notre exemple de reconfiguration.

```
void unloadComp(char *compName);

void loadComp(char *compName,
              char *object);
```

FIG. 6.28 – Interface de communication

La méthode `unloadComp` permet de demander au système d'exploitation de décharger le composant dont le nom est donné en argument. La méthode `loadComp` provoque quant à elle le chargement du composant dont le nom et le binaire sont passés en paramètres. Ces deux méthodes s'appuient sur les mécanismes de reconfiguration fournis par le système pour effectuer le travail requis.

Le noyau reconfigurable Le système d'exploitation s'exécutant sur le routeur actif est basé sur un noyau dynamiquement reconfigurable réalisé à l'aide de l'architecture THINK. Les mécanismes mis en œuvre pour permettre la reconfiguration du système sont présentés en détails dans [Sen03]. Les mécanismes qui nous utilisons ici sont ceux permettant de charger et décharger dynamiquement un composant en mémoire à partir d'un binaire et d'accéder ensuite à ses méthodes via le canevas classique de THINK.

6.7 Conclusion

Ces exemples valident les outils et le canevas sécurisé que nous avons implantés. Ils démontrent qu'il est possible d'introduire de la sécurité dans un système d'exploitation sans sacrifier sa flexibilité.

Le mécanisme d'isolation mémoire logicielle reste complètement neutre vis à vis de la politique d'isolation choisie comme le montre notre exemple de modification dynamique de cette politique qui ne nécessite pas de changer l'outil. L'ordonnanceur de disque adaptable permet de changer dynamiquement l'algorithme d'ordonnancement des requêtes disque en fonction des besoins du système, sans modifier les mécanismes de base de gestion de la file des requêtes. Le gestionnaire réseau sécurisé est un exemple d'outil reconfigurable dynamiquement grâce à ses méthodes de modification de ses différents paramètres. L'exemple de la modification dynamique de la politique d'isolation mémoire illustre de plus la collaboration nécessaire entre l'outil d'isolation qui assure la sécurité au niveau système, le canevas sécurisé qui assure la sécurité au niveau des composants, et le gestionnaire de la politique de sécurité qui les coordonne et les paramètre.

Le canevas de gestion de ressources que nous avons implanté dans l'architecture THINK est un exemple de mise en œuvre des principes concernant le canevas logiciel sécurisé et montre qu'ils peuvent être appliqués pour satisfaire des objectifs dépassant le simple contrôle d'accès, notamment en ce qui concerne la spécification de politiques. Il illustre de plus notre démarche de séparation de la gestion de la politique des outils servant à la mettre en œuvre, comme le montre l'exemple d'un outil de comptabilisation des consommations chaînées de ressources pouvant être utilisé par le gestionnaire de ressources pour protéger le système contre les attaques contre la qualité de service.

Enfin, l'exemple de l'implantation d'un environnement d'exécution sécurisé pour rou-

teurs actifs montre que nos résultats sont utilisables pour des applications réalistes pouvant être implantées rapidement avec l'architecture THINK. On utilise notre canevas sécurisé et l'outil d'isolation mémoire logicielle pour garantir l'exécution sécurisée des programmes contenus dans les paquets actifs. Grâce aux mécanismes de reconfiguration inclus dans le système d'exploitation sous-jacent, nous sommes capables de remplacer dynamiquement le composant gérant le réseau par un composant intégrant nos mécanismes de protection contre les attaques par inondation, illustrant ainsi la bonne flexibilité du système.

Chapitre 7

Conclusion et perspectives

7.1 Synthèse

Le travail que nous avons présenté démontre qu'il est possible de concilier flexibilité et sécurité dans un noyau de système d'exploitation.

Les mécanismes de protection mis en place dans les systèmes classiques sont le plus souvent très peu flexibles du fait de l'intégration de la politique dans le mécanisme, qui oblige le constructeur d'un système à modifier massivement le code du noyau s'il désire changer de politique de protection. De plus, les mécanismes sont généralement peu paramétrables et beaucoup de décisions sont fixées à la compilation, rendant le système très peu évolutif dynamiquement.

Des architectures alternatives ont été proposées pour essayer d'améliorer la flexibilité des systèmes. Cependant, cette amélioration de la flexibilité se fait le plus souvent au détriment d'une autre propriété du système, comme la sécurité ou les performances par exemple. Les projets de recherche conduits dans le domaine des systèmes flexibles mettent en évidence un certain nombre d'idées qu'il nous a paru intéressant de développer et de combiner pour atteindre notre objectif. L'architecture THINK offre une base facilitant la construction de systèmes flexibles. Notre démarche dans ce travail a été de partir de cette base pour y ajouter des mécanismes de sécurité en vérifiant systématiquement qu'on ne réduisait pas la flexibilité originale se faisant.

Les outils élémentaires de protection que nous avons spécifiés et implantés permettent de protéger le système contre des attaques ciblées. À la différence des gestionnaires traditionnellement implantés dans les systèmes classiques, ces outils restent complètement indépendants de la politique de gestion choisie. Cela permet d'assurer leur flexibilité et facilite le travail du programmeur système en regroupant la gestion des politiques dans un composant unique, le gestionnaire de sécurité. Le mécanisme d'isolation mémoire que nous avons implanté est beaucoup plus flexible qu'une isolation matérielle puisque les

contrôles d'accès sont générés dynamiquement par le gestionnaire de sécurité. Le gestionnaire de disque adaptable est complètement flexible vu qu'il n'impose aucun algorithme d'ordonnancement et fournit au programmeur les outils dont il a besoin pour évaluer le comportement de l'algorithme en place et le remplacer si besoin est. Enfin le gestionnaire de réseau permet de paramétrer à sa guise les outils de protection contres les attaques en délocalisant toutes les prises de décision dans le gestionnaire de sécurité.

La sécurisation du canevas logiciel de THINK montre qu'il est possible de mettre en place un modèle de programmation à la fois flexible et sûr dans un système. Cette sécurisation a nécessité la modification de l'implantation originale du canevas logiciel de THINK car la sécurité n'avait pas été prise en compte lors de son développement. De façon plus générale on peut appliquer une sécurisation similaire dans tous les systèmes imposant un modèle de programmation pour les modules composant le noyau. En contrôlant l'enregistrement des modules et leurs interactions, ce qui dans THINK revient à contrôler l'exportation des interfaces, la création de liaisons et leur utilisation lors des appels de méthodes, on peut mettre en place un contrôle d'accès au niveau applicatif qui est complémentaire du contrôle d'accès bas niveau assuré par l'isolation mémoire. L'ajout d'un module gestionnaire de sécurité regroupant toutes les prises de décision ne pose pas de problème dans la majorité des systèmes et permet, couplé avec des outils élémentaires de protection indépendants de la politique mise en œuvre, d'assurer la flexibilité de la gestion de la sécurité dans le système.

Les différentes évaluations que nous avons réalisées permettent de vérifier la flexibilité des mécanismes de protection proposés. On démontre la flexibilité du mécanisme d'isolation mémoire logicielle en modifiant dynamiquement la politique d'isolation sans avoir à modifier ni l'outil ni le code généré grâce à la délocalisation des capacités matérialisant la politique de protection dans le gestionnaire de sécurité.

De même on prouve qu'on peut remplacer dynamiquement l'algorithme d'ordonnancement des requêtes disque sans avoir à modifier l'outil qui gère la file des requêtes et effectue les accès au disque. Comme l'illustre l'exemple choisi, un tel changement dynamique de la politique d'ordonnancement des requêtes disques peut être nécessaire en cas d'attaque contre la qualité de service du système, mais il peut également être utile pour adapter le système à un type particulier d'applications, par exemple des applications multimédia nécessitant des garanties sur les temps d'accès disque afin d'assurer une qualité vidéo optimale. Un tel outil, en plus de pouvoir être utile pour se protéger contre les attaques par déni de service, améliore donc la capacité d'adaptation globale du système.

Le gestionnaire de réseau peut quant à lui être paramétré de façon à adapter le degré de protection du système en fonction des besoins. Ainsi par exemple, un système servant à la fois de serveur HTTP et de machine de travail pour des utilisateurs locaux devra garantir que le service fourni aux utilisateurs ne sera pas perturbé en cas d'attaque par inondation

par exemple, et donc paramétrer le gestionnaire de réseau pour rejeter toute possibilité d'attaque, même si au final cela se traduit par des pertes de connexions légitimes. A contrario une machine se consacrant uniquement à la gestion des requêtes HTTP pourra être paramétrée de façon à optimiser le service Internet fourni.

Le canevas de gestion de ressources que nous avons détaillé est un exemple simple d'un canevas de protection dépassant le strict contrôle d'accès. Il illustre la mise en œuvre des principes présentés pour le canevas sécurisé de THINK, notamment la définition de politiques de gestion de ressources qui est très similaire à la spécification de politiques de contrôle d'accès en Prolog. Il montre aussi comment on peut, en modifiant très légèrement le canevas sécurisé, mettre en place un comptage chaîné des consommations de ressources, un outil important pour protéger le système contre les attaques par déni de service.

Enfin, l'application de nos résultats au domaine des réseaux actifs montre qu'ils sont applicables dans un exemple réaliste de système. La construction d'un environnement d'exécution sécurisé pour paquets actifs met en œuvre nos outils élémentaires (ici le mécanisme d'isolation logicielle) et le canevas sécurisé que nous avons implanté dans THINK. Combinés avec des mécanismes de reconfiguration dynamique du système, nos mécanismes permettent la construction d'un système d'exploitation flexible et sûr, ce qui était notre objectif de départ.

7.2 Perspectives

Il reste à l'issue de ce travail plusieurs points sur lesquels nous pensons qu'un travail complémentaire serait intéressant.

7.2.1 Gestion des politiques de sécurité

Comme nous l'avons indiqué, la spécification et la gestion dynamique de politiques de sécurité est un travail complexe qui dépasse le cadre du travail présenté ici. Si de nombreux travaux ont été conduits concernant la gestion de politiques d'administration dans les systèmes distribués à grande échelle, peu de résultats ont été obtenus dans le cadre particulier des noyaux de systèmes d'exploitation. Les contraintes sont en effet beaucoup plus fortes en ce qui concerne les performances puisque dans un système distribué certains coûts de résolution de requêtes peuvent être « noyés » dans les coûts des communications via le réseau ce qui n'est pas le cas évidemment dans un noyau.

Nous avons proposé un outil permettant de traduire des politiques spécifiées statiquement en Prolog vers du code C adapté au canevas sécurisé de THINK. Il peut être intéressant d'essayer de généraliser cet outil pour le rendre plus indépendant de l'implantation du canevas logiciel du système. Il est également possible d'intégrer un outil similaire dans un composant chargeur de politiques, dont le rôle serait d'interpréter les politiques

exprimées en Prolog pour affecter dynamiquement les champs correspondants dans la structure représentant les capacités (ou une autre structure appropriée si un mécanisme de contrôle d'accès différent est choisi). Le problème de cette approche reste cependant qu'il n'est pas possible d'ajouter des règles dans un format non prévu lors de la construction du système. Dans l'exemple de notre gestion des droits d'accès sous forme de capacités, il n'est pas possible d'ajouter dans la structure gérant les capacités des champs différents des droits d'exportation, de liaison et d'appels.

Une autre possibilité est de définir un langage intermédiaire pouvant être interprété plus rapidement que Prolog par le gestionnaire de sécurité. Ce langage doit de plus permettre l'ajout et la modification dynamique de règles de sécurité. Un tel langage, qui reste à spécifier, serait un langage spécifique au domaine de la protection tel que défini dans [VDKV00].

7.2.2 Spécification d'une architecture de gestion de la qualité de service

Le canevas de gestion de ressources que nous avons présenté est un exemple simple illustrant la méthode mise en œuvre pour sécuriser le canevas logiciel de THINK. Il serait intéressant de développer dans l'architecture THINK un canevas de gestion de ressources complet permettant d'assurer de façon flexible la qualité de service dans le système généré. En plus de la définition d'un langage de spécification de politiques de gestion de ressources qui s'apparentera vraisemblablement à celui concernant la spécification de politiques de sécurité, il peut être intéressant de fournir des outils facilitant les tâches d'administration du système.

Par exemple, notre outil de comptage des consommations chaînées de ressources pourrait être utilisé par un mécanisme capable d'établir des clichés de la consommation de chaque ressource dans le système à un instant donné sous la forme d'une console de surveillance. Cet outil de surveillance pourrait également être capable de générer des journaux de l'évolution des allocations et libération de ressources, qui pourraient être utiles dans le cas d'une attaque par déni de service aboutissant à un arrêt du système, pour effectuer une analyse post-mortem des causes de la panne.

Un autre point important dans la gestion de la qualité de service est la possibilité de définir par avance un plan d'allocation des ressources, qui garantira que les ressources réservées à une application seront bien disponible lorsqu'elle en aura besoin. De telles réservations de ressources peuvent par exemple être mises en œuvre grâce à des quotas flexibles similaires à ceux présentés dans [LNZO02] pour la gestion de l'espace disque.

Enfin, un point essentiel à vérifier est le coût des mécanismes mis en œuvre, et particulièrement des mécanismes de comptage qui sont potentiellement les plus fréquemment utilisés. En effet, il ne faudrait pas que l'exécution du système et des applications soit pénalisée par l'ajout des mécanismes servant à la gestion. De plus, il est important de

prendre en compte le coût de la gestion de ressources dans le comptage des ressources consommées dans le système, à défaut de quoi ce comptage sera imprécis.

Un tel canevas serait particulièrement intéressant dans le cadre du portage de THINK sur des systèmes embarqués où la gestion des ressources est primordiale vues les quantités disponibles souvent très limitées. Un travail est en cours concernant le prototypage d'une version temps réel de l'architecture THINK [Cha03] dans laquelle un tel canevas pourrait notamment être utilisé pour la gestion de contraintes temps réel.

Liste de publications

Nous présentons ci-dessous la liste des publications attenantes à cette thèse. Toutes ces publications peuvent être téléchargées sur le site Internet du projet Sardes à l'adresse : `http://sardes.inrialpes.fr/papers/`.

Revues internationales

2003

Christophe Rippert. Protection in Flexible Operating System Architectures. In *ACM SIGOPS Operating System Review*, volume 37, number 4, pages 8-18, October 2003.

Conférences internationales

2002

Christophe Rippert and Jean-Bernard Stefani. THINK : A Secure Distributed Systems Architecture. In *Proceedings of the 10th ACM SIGOPS European Workshop*, St Emilion (France), September 22nd-25th, 2002.

Christophe Rippert and Jean-Bernard Stefani. Building secure embedded kernels with the Think architecture. In *Proceedings of the workshop on Engineering Context-aware Object-Oriented Systems and Environments, in association with OOPSLA'2002*, Seattle (USA), November 4th-8th, 2002.

Christophe Rippert. Component isolation in the Think architecture. In *Proceedings of the 7th CaberNet Radicals workshop*, Bertinoro (Italy), October 13th-16th, 2002.

2001

Christophe Rippert and Jean-Bernard Stefani. Protection in the Think exokernel. In *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems*, Bertinoro (Italy), May 14th-18th, 2001.

Christophe Rippert and Daniel Hagimont. An evaluation of the Java Card environment. In *Proceedings of the Advanced Topic Workshop Middleware for Mobile Computing*, Heidelberg (Germany), November 16th, 2001.

Conférences nationales

2002

Christophe Rippert et Jean-Bernard Stefani. Éléments de sécurité dans l'architecture de systèmes répartis THINK. In *Proceedings of ASF 2002, Journées francophones des jeunes chercheurs en systèmes d'exploitation*, Hammamet (Tunisie), 10-13 Avril 2002.

Bibliographie

[ABB+86] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach : A new kernel foundation for Unix development. In *Proceedings of the Summer USENIX 1986 Technical Conference*, 1986.

[Aiv02] T. Aivazian. Linux Kernel 2.4 Internals, 2002. `http ://www.tldp.org/LDP/lki/index.html`.

[BALL89] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight Remote Procedure Call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989.

[BC00] G. Bianchi and A. Campbell. A Programmable MAC Framework for Utility-Based Adaptive Quality of Service Support. *IEEE Journal of Selected Areas in Communications*, 18(2), 2000.

[BCE+94] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN — An Extensible Microkernel for Application-specific Operating System Services. In *Proceedings of the 6th ACM SIGOPS European Workshop*, 1994.

[BL73] D. Bell and L. LaPadula. Secure computer systems : Mathematical foundations and model. Technical Report M74-244, MITRE Corperation, 1973.

[Bro95] K. Brockshmidt. *Inside OLE.* Microsoft Press, seconde edition, 1995.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM symposium on Operating Systems Principles*, 1995.

[BSS01] E. Bott, C. Siechert, and C. Stinson. *Microsoft Windows XP Inside Out.* Microsoft Press, 2001.

[Ca98] K. Calvert and al. Architectural Framework for Active Networks. AN Architecture Working Group, 1998.

[CC98] M. Clarke and G. Coulson. An Architecture for Dynamically Extensible Operating Systems. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, 1998.

[Cha03] O. Charra. *Conception de noyaux de systèmes embarqués reconfigurables*. Thèse de Doctorat. Université Joseph Fourier — Grenoble 1, 2003.

[CJ75] E. Cohen and D. Jefferson. Protection in the Hydra Operating System. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, 1975.

[CV65] F.J. Corbató and V.A. Vyssotsky. Introduction and Overview of the Multics System. In *Proceedings of the AFIPS Fall Join Computer Conference*, 1965.

[DAR81a] DARPA. Internet Protocol. Request for comments : 792, Network Working Group, 1981.

[DAR81b] DARPA. Transmission Control Protocol. Request for comments : 793, Network Working Group, 1981.

[DAR81c] DARPA. T/TCP — TCP Extensions for Transactions. Request for comments : 793, Network Working Group, 1981.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog : The Standard*. Springer, 1996.

[DPP+97] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. W. Trickey, and Ph. Winterbottom. The Inferno Operating System. Technical Report 1, volume 2, Bell Labs, 1997.

[DVH66] J. B. Dennis and E. C. Van Horn. Programming Semantics For Multiprogrammed Computations. Technical Report MIT/LCS/TR-23, Massachusetts Institute of Technology, 1966.

[EK95] D. R. Engler and M. F. Kaashoek. Exterminate All Operating System Abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, 1995.

[EKO95] D. R. Engler, M. F. Kasshoek, and J. O'Toole. Exokernel : An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[Fas01] J.-Ph. Fassino. *THINK : vers une architecture de systèmes flexibles*. Thèse de Doctorat. École Nationale Supérieure des Télécommunications, 2001.

[FBB+97] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit : A Substrate for Kernel and Language Research. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[FSLM02] J.-Ph. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK : A Software Framework for Component-based Operating System Kernels. In *Proceedings of the Usenix Annual Technical Conference*, 2002.

[Gie90] M. Gien. Micro-kernel Architecture : Key to Modern Operating Systems Design. Technical Report CS/TR-90-42.1, Chorus Systems, 1990.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Langage Specification*. Addison-Wesley, seconde edition, 2000.

[Gra67] R. M. Graham. Protection in an Information Processing Utility. In *Proceedings of the 1st ACM Symposium on Operating Systems Principles*, 1967.

[GSB+99] E. Gabber, Ch. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *Proceedings of the USENIX Annual Technical Conference*, 1999.

[Hag98] D. Hagimont. *Accès à l'information répartie : adressage et protection*. Habilitation à Diriger des Recherches, Institut National Polytechnique de Grenoble, 1998.

[Har92] S. P. Harbison. *Modula-3*. Prentice Hall, 1992.

[HF98] J. Helander and A. Forin. MMLite : A Highly Componentized System Architecture. In *Proceedings of the 8th ACM SIGOPS European Workshop*, 1998.

[Hil92] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

[HK99] M. Hicks and A. D. Keromytis. A Secure PLAN. In *Proceedings of the First International Working Conference on Active Networks*, 1999.

[Int97] Intel Architecture Software Developer's Manuel, 1997. `http ://www.intel.com`.

[Jen99] Ch. D. Jensen. *Un modèle de contrôle d'accès générique et sa réalisation dans la mémoire virtuelle répartie unique Arias*. Thèse de Doctorat. Université Joseph Fourier — Grenoble 1, 1999.

[KS74] P. A. Karger and R. R. Schell. MULTICS Security Evaluation : Vulnerability Analysis. Technical Report ESD-TR-74-193 Vol. II, Electronic Systems Division - Air Force Safety Center, 1974.

[KS02] P. A. Karger and R. R. Schell. Thirty Years Later : Lessons from the Multics Security Evaluation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, 2002.

[Lam71] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, 1971.

[LCC+75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, 1975.

[Lie96] J. Liedtke. Toward Real Microkernels. *Communications of the ACM*, 39(9) :70–77, 1996.

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal of Selected Areas in Communications*, 14(7) :1280–1297, 1996.

[LMSY95] E. Lupu, D. Marriott, M. Sloman, and N. Yialelis. A Policy Based Role Framework for Access Control. In *Proceedings of the 1st ACM/NIST Workshop on Role-Based Access Control*, 1995.

[LNZO02] O. C. Leonard, J. Nieh, E. Zadok, and J. Osborn. The Design and Implementation of Elastic Quotas : A System for Flexible File System Management. Technical Report CUCS-014-02, Columbia University, 2002.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, seconde edition, 1999.

[MM00] J. Mauro and R. McDougall. *Solaris Internals : Core Kernel Architecture*. première edition, 2000.

[MP96] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[MP00] M. Margaritidis and G. Polyzos. MobiWeb : Enabling adaptive continuous media applications over wireless links. In *Proceedings of the IEEE International Conference on Third Generation Wireless Communications*, 2000.

[MS93] J. Moffett and M. Sloman. Policy Hierarchies for Distributed Systems Management. *IEEE Journal on Selected Areas in Communications*, 11(9) :1404–1414, 1993.

[MVRT+90] S. J. Mullender, G. Van Rossum, A. S. Tanenbaum, R. Van Renesse, and H. Van Staveren. Amoeba : A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5) :44–53, 1990.

[ODP] ODP Reference Model, Foundations, ITU-T ISO/IEC Recommendation X.902 International Standard 10746-2, 1995.

[OFSS96] D. Olawsky, T. Fine, E. Schneider, and R. Spencer. Developing and using a "Policy Neutral" Access Control Policy. In *Proceedings of the New Security Paradigms Workshop*, 1996.

[Pos81] J. Postel. Internet Control Message Protocol. Request for comments : 792, Network Working Group, 1981.

[PR85] J. Postel and J. Reynolds. File Transfer Protocol (FTP). Request for comments : 959, Network Working Group, 1985.

[RFS+00] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit : Component Composition for Systems Software. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[Rip99] Ch. Rippert. *Projet JCOD : Compilation à la demande d'applications Java pour systèmes embarqués.* Rapport de Magistère. Unité de Formation et de Recherche en Informatique et Mathématiques Appliquées, Université Joseph Fourier — Grenoble 1, 1999.

[RT74] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7) :365–375, 1974.

[Sal74] J. H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communication of the ACM*, 17(7), 1974.

[Sen03] A. Senart. *Canevas logiciel pour la construction de noyaux de systèmes d'exploitations dynamiquement adaptables.* Thèse de Doctorat. Institut National Polytechnique de Grenoble, 2003.

[SESS96] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster : Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[Slo94] M. Sloman. Policy Driven Management For Distributed Systems. 2(4) :333–360, 1994.

[Soc99] The Internet Society. Hypertext Transfer Protocol – HTTP/1.1. Request for comments : 2616, Network Working Group, 1999.

[SP99] O. Spatscheck and L. L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[SS98] D. Sisalem and H. Schulzrinne. The loss-delay adjustment algorithm : a TCP-friendly adaptation scheme, 1998.

[VDKV00] A. Van Deursen, P. Klint, and J. Visser. Domain-Specific Languages — An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, 2000.

[VH96] A. C. Veitch and N. C Hutchinson. Kea — A Dynamically Extensible and Configurable Operating System Kernel. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, 1996.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5) :203–216, 1993.

[WLP75] W. Wulf, R. Levin, and C. Pierson. Overview of the Hydra Operating System development. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, 1975.

[Zim95] Ph. R. Zimmermann. *PGP : Source Code and Internals*. MIT Press, 1995.