Sécurité et processus mobiles

Sylvain HUET

juin 1996

Contents

Le présent mémoire rend compte du stage que j'ai effectué de mars à juin 1996, dans le cadre du diplôme d'études approfondies d'algorithmique, dirigé par Robert Cori, professeur à l'Ecole Polytechnique, et avec l'accord de Jacques Stern, responsable de la filière Cryptographie, et professeur à l'Ecole Normale Supérieure de la rue d'Ulm.

Ce stage s'est déroulé sous la direction et au sein de l'équipe de Jean-Jacques Lévy, responsable du projet PARA à l'unité Rocquencourt de l'Institut National de Recherche en Informatique et en Automatique.

Chapter 1

Présentation générale

1.1 Les processus mobiles dans l'évolution de l'informatique

Les premiers ordinateurs, tels l'Eniac, avaient pour particularité de ne pas avoir de programme en mémoire. Leur programmation se faisait en modifiant manuellement leurs circuits, à savoir en connectant différemment les différents modules qui les composaient. C'était donc un programme inscrit en dur, directement dans l'électronique de la machine.

L'aspect peu pratique d'un tel fonctionnement a rapidement abouti à la réalisation de machines où le programme était représenté sous la forme d'une suite de nombres stockés dans la mémoire de l'ordinateur. La suite de ces nombres forme ce que l'on appelle le *code* d'un programme.

L'intérêt fondamental de ce schéma est la possibilité de dupliquer rapidement un programme d'une machine vers une autre : il suffit de recopier le code sur une nouvelle machine, via par exemple une bande magnétique, un disque magnétique ou optique, ou encore un réseau.

Pendant longtemps les machines sont restées relativement isolées. Les transmissions de données par téléphone étaient lentes, et les autres types de réseaux avaient un développement géographique faible. La transmission de codes programme ne pouvait s'envisager de manière massive, mais seulement de manière ponctuelle. Le développement réçent du réseau *internet* ouvre d'autres perspectives.

D'une part, l'extension mondiale du réseau et l'améliation annoncée de sa bande passante banalisent la notion de communication numérique entre machines. D'autre part, le développement du *Web* modifie le caractère des communications qui passent d'un mode convenu, où les deux interlocuteurs se mettent d'accord et fixent par exemple un rendez-vous pour la transmission, à un mode de publication dans lequel un utilisateur du réseau peut mettre à la disposition du public un service particulier.

Le Web permet typiquement la diffusion de textes, d'images et de sons, présentés à celui qui les consulte au travers d'une interface standard. Un évolution actuelle consiste à ajouter à la liste précédente la possibilité de diffuser du code programme, dont l'utilité est par exemple de modifier l'interface de visualisation. Il s'agit alors de *code mobile* : les exemples en sont le langage Java, développé par Sun, et intégré dans le browser de Netscape, et le langage Caml, intégré dans le browser MMM développé à l'INRIA par F.Rouaix. Il s'agit d'importer du code, via le réseau, et de l'exécuter dans un contexte particulier, dont le rôle est de rendre insensibles les différences entre machines. Ce programme est alors capable d'accéder à certaines ressources de la machine, typiquement l'écran, une partie de la mémoire de masse, et le réseau.

L'étape suivante consiste à déplacer non plus le code du programme, mais le programme lui-même, en cours d'exécution. On parle alors de *processus mobile*. La différence importante entre les notions de code mobile et de processus mobile est la suivante : le code mobile est figé, c'est-à-dire que la machine qui l'importe le reçoit tel qu'il a été créé par son auteur, et que son exécution est reproductible. En revanche, lorsqu'un processus mobile est accueilli par une machine, il a généralement déjà visité d'autres machines, avec lesquelles, via le réseau, il a pu garder des liens. On notera que la notion de code mobile est contenue dans celle de processus mobile.

1.2 Rôle de la mobilité

Comme nous le détaillerons plus loin, l'équipe de Jean-Jacques Lévy à conçu un calcul des processus mobiles, appelé *Join-calcul* (prononcer à l'anglaise), dérivé du π-calcul. Ce calcul offre une indépendance presque complète des possibilités d'un processus vis-à-vis de sa position géographique. La seule exception concerne la modélisation des pannes : en effet, la vie d'un processus ne dépend que de celle de la machine qui l'abrite ; quand un processus se déplace, sa vulnérabilité change, sans nécessairement augmenter ou dimin-

uer.

Dans ces conditions, si un processus a exactement les mêmes possibilités quelle que soit sa position, la question de l'intérêt de la mobilité se pose de manière pertinente.

La réponse la plus commune concerne les performances. D'une part, la mobilité permet de répartir la charge de travail sur différentes machines. Par exemple, on souhaite trouver la décomposition d'un grand nombre entier : un processus chargé de se travail va se dupliquer et se déplacer sur un grand nombre de machines, chaque sous-processus effectuera une partie de la recherche, puis communiquera son résultat à la machine d'origine. D'autre part, si un processus souhaite accéder de manière intensive à une ressource, telle qu'un écran ou un disque, ses performances seront d'autant meilleures qu'il sera géographiquement proche de la ressource.

1.3 Questions de sécurité

La sécurité des systèmes informatiques est une notion qui remonte à l'apparition des systèmes multi-utilisateurs, sur lesquels un ensemble d'utilisateurs doivent partager des ressources communes mais disposent de ressources (typiquement sous forme d'espace disque) qui leur sont réservées et dont ils contrôlent les accès.

Typiquement, la sécurité amène à considérer l'informatique comme un ensemble de ressources auxquelles accèdent un ensemble d'utilisateurs. Tout dans la machine est ressource : mémoire, temps machine, disque, réseau, écran, clavier, etc... Les utilisateurs sont une notion plus humaine, qui n'apparaît qu'au niveau du logiciel. La sécurité consiste à définir des liens entre les utilisateurs et les ressources, liens correspondant à des droits d'accès. La notion de personne est assez vague en informatique, puisque l'être humain communique à travers des périphériques. L'utilisateur est donc représenté en informatique par le ou les processus qu'il contrôle et qui le représentent.

Listes d'accès

Le problème de la sécurité peut être vu sous différents angles. La première approche passe par la définition de l'identité d'un processus. Le principe le plus répandu consiste à associer à chaque processus le nom d'un util-

isateur, ainsi que souvent des noms de groupes ou de projets. L'accès à une ressource se fera au nom de ces différentes identités. La gestion des identités des processus est typiquement assurée par le système. Ce système est connu sous le nom de *listes d'accès*, car la ressource gère typiquement une liste d'identités autorisées.

Les difficultés apparaissent lorsqu'un utilisateur souhaite utiliser un programme qu'il n'a pas écrit, ce qui est évidemment le cas le plus fréquent. Ce programme doit-il être exécuté avec les droits de l'auteur ou avec celui de l'utilisateur ? Il n'y a pas de réponse simple ou même élegante à cette question. L'auteur ne sait pas quel usage va être fait de son programme, et n'a pas a priori les droits d'accéder aux ressources de l'utilisateur. Or, si le programme est un outil, par exemple un traitement de texte, l'utilisateur souhaite l'utiliser pour lire et modifier ses propres fichiers, non ceux de l'auteur. Ceci pousse à penser que le programme doit s'exécuter avec les droits de l'utilisateur. Mais inversement, celui-ci ne sait pas vraiment ce que contient le programme qu'on lui présente. Il ne sait pas si le traitement de texte ne va pas effacer certains de ses fichiers (de manière malveillante, ou à cause d'une erreur de programmation), ou lire des informations confidentielles de l'utilisateur et les transmettre par le réseau vers un tiers. Ce sont là des arguments pour que le programme s'exécute avec les droits de l'auteur, sans quoi l'utilisateur doit faire confiance à l'auteur.

Capacités

La seconde approche consiste à distribuer des *tickets au porteur*. Chaque ticket correspond à un certain type d'accès à une certaine ressource. Lorsqu'une ressource est créée, au moins un ticket est produit, et fourni à l'utilisateur qui a demandé la création de la ressource. L'accès à une ressource est autorisé à condition que le ticket correspondant soit montré. Il n'y a pas de notion d'identité dans ce schéma. Les tickets peuvent être dupliqués, échangés, etc... Un tel système est appelé *système à capacités*. C'est a priori un moyen intéressant de régler les problèmes de sécurité dans les milieux fortement distribués, puisqu'il ne requiert pas de gestion d'identité.

Le problème des capacités est justement leur diffusion. Si la ressource X donne une capacité c à A et à B, il lui est impossible de retirer la capacité c à A sans la retirer à B. Au mieux X doit invalider la capacité c (c'est-à-dire ne plus la reconnaître comme valide) et distribuer une nouvelle capacité équivalente c' à B seul. Cependant, rien n'empêche alors B de décider

unilatéralement de la communiquer à A.

Spécificité des processus mobiles

La solution adoptée par le système Unix est de type liste d'accès. A chaque ressource est affectée une liste d'identités ayant des droits de lecture, d'écriture ou d'exécution. Lorsqu'un programme est exécuté, il prend les droits de l'utilisateur ou du propriétaire (c'est le propriétaire qui le décide). La notion de propriétaire se rapproche de celle d'auteur, mais reste différente. Les utilisateurs et les propriétaires sont des utilisateurs d'une même machine, et sont connus et contrôlables.

Dans un système de code mobile, le propriétaire du code, que l'on peut appeler le diffuseur, et l'utilisateur peuvent se trouver sur des machines qui n'ont aucun point commun, notamment en ce qui concerne l'administration. Les droits de l'utilisateur et du diffuseur s'exercent dans des domaines différents et ne sont donc pas comparables. Puisque le code s'exécute sur la machine de l'utilisateur, c'est avec les droits de l'utilisateur qu'il s'exécute. En cas de disfonctionnement du code, ou de code malveillant, il est très difficile à l'utilisateur de demander des comptes au diffuseur, alors qu'il est simple de demander des explications à un propriétaire sous Unix. Le système de sécurité adopté pour les codes mobiles consiste donc à installer une couche de sécurité entre le code et les ressources, de manière à pouvoir exécuter le code avec un sous-ensemble des droits de l'utilisateur, puis à demander à celui-ci de fixer les limites de ce sous-ensemble, en fonction de paramètres tels que la provenance du code (adresse du diffuseur), ou qu'une signature électronique authentifiant l'auteur du code.

Avec les processus mobiles, le problème est encore différent. Autant le code mobile se limite au rôle d'outils ou d'interfaces que l'on peut distribuer, autant les processus mobiles offrent de nouvelles perspectives, comme celle d'agents se déplaçant à travers le réseau, et conservant des liens vers leur site d'origine. La frontière entre ces deux types de processus n'est pas nette, et indépendamment des questions de sécurité, il n'est pas possible de dire, dans le cas général, pour qui travaille un processus. Pourtant, ne serait-ce qu'en cas de disfonctionnement, il faut savoir vers qui l'administration d'un système peut se retourner. De plus le processus mobile présente une originalité importante : il peut changer *d'employeur*, c'est-à-dire travailler au service d'un processus, qui le cède à un autre et ainsi de suite.

Chapter 2

π-calcul et Join-calcul

2.1 Le π-calcul. Milner-Parrow-Walker

Le π-calcul est un calcul modélisant les communications entre processus.

2.1.1 Le π-calcul monadique

x, y, z	$\in$	X	noms
P, Q	::==		processus
	|	$\sum_{i \in I} \pi_i.P_i$	somme (I fini)
	|	$P \vert Q$	parallèle
	|	$!P$	itération
	|	$(\nu x)P$	restriction
π	::==		action atomique
	|	$x(y)$	input sur y le long de x
	|	$\overline{x}y$	output y le long de x

2.1.2 Le π-calcul polyadique

Le π-calcul polyadique se code de la manière suivante :

$$\begin{aligned} x(y_1 \dots y_n) &= x(w).w(y_1) \dots w(y_n) \\ \overline{x}y_1 \dots y_n &= (\nu w)\overline{x}w.\overline{w}y_1 \dots \overline{w}y_n \end{aligned}$$

Un simple codage $x(y_1 \dots y_n) = x(y_1) \dots x(y_n)$ serait en effet incorrect lors d'utilisations concurrentes du canal x.

2.1.3 Liaison des variables

- Si $I = \emptyset$, on écrit $0 = \sum_{i \in I} \pi_i.P_i$,
- y est une variable liée dans $x(y).P$,
- y est une variable libre dans $\overline{x}y.P$,
- x est une variable libre dans $x(y).P$ et $\overline{x}y.P$,
- x est une variable liée dans $(\nu x)P$,
- x est le sujet et y l'objet dans $x(y).P$, et $\overline{x}y.P$,
- on écrit π pour $\pi.0$

2.1.4 Equivalence structurelle

- $P \equiv Q$ par α-conversion (renommage des variables liées)
- $!P \equiv P|!P$
- $P|Q \equiv Q|P$
- $P|0 \equiv P$
- $P|(Q|R) \equiv (P|Q)|R$
- $P + Q \equiv Q + P$
- $P + 0 \equiv P$
- $P + (Q + R) \equiv (P + Q) + R$
- $(\nu x)0 \equiv 0$
- $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
- $(\nu x)(P|Q) \equiv P|(\nu x)Q$ quand x non libre dans P

2.1.5 Règles de réduction

comm : $(R + x(y).P)|(R' + \overline{x}z.Q) \rightarrow P\{z/y\}|Q$

par : $\frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$

res : $\frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$

struct : $\frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$

Il est à noter que le π-calcul n'est pas déterministe.
$x(u).\overline{y}(u) \mid \overline{x}(a) \mid \overline{x}(b)$ se réduit :

- soit en $\overline{y}(a) \mid \overline{x}(b)$
- soit en $\overline{y}(b) \mid \overline{x}(a)$

La somme peut également présenter de l'indéterminisme :
$(x(u).\overline{y}(u) + x(u).\overline{z}(u)) \mid \overline{x}(a)$ se réduit :

- soit en $\overline{y}(a)$
- soit en $\overline{z}(a)$

2.1.6 Exemples

Il est possible de coder les entiers en π-calcul, de manière comparable aux entiers de Church du λ-calcul.

$$\underline{n}(xz) \stackrel{\text{def}}{=} \underbrace{\overline{x}.\overline{x}...\overline{x}}_{n \text{ fois}}.\overline{z}$$

Pour définir la somme, on utilise deux expressions :

$$\begin{aligned} Copy(xz, yw) &\stackrel{\text{def}}{=} x.Succ(xz, yw) + z.\overline{w} \\ Succ(xz, yw) &\stackrel{\text{def}}{=} \overline{y}.Copy(xz, yw) \end{aligned}$$

On peut montrer que :

$$\begin{aligned} (\nu xz)(\underline{n}(xz) \mid Copy(xz, yw)) &\approx \underline{n}(yw) \\ (\nu xz)(\underline{n}(xz) \mid Succ(xz, yw)) &\approx \underline{n+1}(yw) \end{aligned}$$

Et on obtient :
$Add(x_1z_1, x_2z_2, yw) \stackrel{\text{def}}{=} x_1.\overline{y}.Add(x_1z_1, x_2z_2, yw) + z_1.Copy(x_2z_2, yw)$

On montre alors que :
$(\nu x_1z_1x_2z_2)(\underline{n_1}(x_1z_1) \mid \underline{n_2}(x_2z_2) \mid Add(x_1z_1, x_2z_2, yw)) \approx \underline{n_1 + n_2}(yw)$

De même on peut coder les booléens :

$$\begin{array}{rcl} [True](b) & = & b(tf).\overline{t} \\ [False](b) & = & b(tf).\overline{f} \\ [\underline{if}\ b\ \underline{then}\ p\ \underline{else}\ q] & = & (\nu t)(\nu f)\overline{b}\ tf.(t.[p] + f.[q]) \end{array}$$

De plus, il est possible de coder le λ-calcul en π-calcul.

2.2 Le Join-calcul

Le join-calcul est un calcul développé pour modéliser les migrations. Il est aussi expressif que le π-calcul asynchrone, et se révèle plus proche d'un langage de programmation.

2.2.1 Notion de machine chimique

Le modèle de machine chimique permet de représenter simplement les réductions modulo les équivalences. Une telle machine est représentée d'une part par une *solution* contenant des *molécules*, et d'autre part par un ensemble de *réactions* chimiques pouvant être éventuellement réversibles. On la note : $D \vdash P$ où D est l'ensemble des réactions, et P celui des molécules.

2.2.2 Syntaxe du join-calcul

Les termes du calcul sont des processus et des définitions :

$$\begin{array}{rcl} P & \stackrel{\text{def}}{=} & x\langle\tilde{v}\rangle \mid \textbf{def}\ D\ \textbf{in}\ P \mid P|P \\ D & \stackrel{\text{def}}{=} & J \rhd P \mid D \wedge D \\ J & \stackrel{\text{def}}{=} & x\langle\tilde{v}\rangle \mid J|J \end{array}$$

Quelques notions pour fixer les idées :

- $x\langle\tilde{v}\rangle$ correspond à un message $\tilde{v}$ envoyé sur le canal x.

- **def** D **in** P permet de définir de nouveaux canaux, et les règles qui leur sont associées.
- $J \rhd P$ exprime une réduction, ou réaction chimique.

2.2.3 Règles du join-calcul

$$
\begin{array}{lrcll}
\textbf{str-join} & \vdash P_1 | P_2 & \rightleftharpoons & \vdash P_1, P_2 & \\
\textbf{str-and} & D_1 \wedge D_2 \vdash & \rightleftharpoons & D_1, D_2 \vdash & \\
\textbf{str-join} & \vdash \textbf{def}\ D\ \textbf{in}\ P & \rightleftharpoons & D_{\sigma_{\mathrm{dv}}} \vdash P_{\sigma_{\mathrm{dv}}} & (\mathit{range}(\sigma_{\mathrm{dv}})\text{nouveau}) \\
\textbf{red} & J \rhd P \vdash J_{\sigma_{\mathrm{rv}}} & \longrightarrow & J \rhd P \vdash P_{\sigma_{\mathrm{rv}}} &
\end{array}
$$

2.2.4 Règles pour le calcul synchrone

$$
\begin{array}{rcl}
\mathrm{x}\langle \tilde{v} \rangle & \stackrel{\text{def}}{=} & x\langle \tilde{v}, k_{\mathrm{X}} \rangle \\
\textbf{reply}\ \tilde{v}\ \textbf{to}\ \mathrm{x} & \stackrel{\text{def}}{=} & k_{\mathrm{X}}\langle \tilde{v} \rangle \\
\mathrm{x}\langle \tilde{v} \rangle; P & \stackrel{\text{def}}{=} & \textbf{def}\ k_{\mathrm{X}}\langle\rangle \rhd P\ \textbf{in}\ \mathrm{x}\langle \tilde{v} \rangle \\
\textbf{let}\ \tilde{u} = \mathrm{x}\langle \tilde{v} \rangle\ \textbf{in}\ P & \stackrel{\text{def}}{=} & \textbf{def}\ k_{\mathrm{X}}\langle \tilde{u} \rangle \rhd P\ \textbf{in}\ \mathrm{x}\langle \tilde{v} \rangle
\end{array}
$$

Le calcul synchrone est une manière de définir l'ordre dans lequel des réductions doivent s'effectuer. Pour fixer l'antériorité de l'étape $E1$ sur l'étape $E2$, il suffit de créer un canal k ; l'étape $E1$ se termine en écrivant sur ce canal k, tandis que l'étape $E2$ commence par lire sur le canal k.

2.2.5 Exemple

Le programme suivant permet de définir une case mémoire, accessible par des fonctions *get* et *set*.

$$
\textbf{def}\ \mathit{mkcell}\langle v_0, k_0 \rangle \rhd \left(\begin{array}{rl} \textbf{def} & \mathit{get}\langle k \rangle | s\langle v \rangle \rhd k\langle v \rangle | s\langle v \rangle \\ \wedge & \mathit{set}\langle u, k \rangle | s\langle v \rangle \rhd k\langle\rangle | s\langle u \rangle \\ \textbf{in} & s\langle v_0 \rangle | k_0\langle \mathit{get,set} \rangle \end{array} \right)
$$

La présence d'un message $\mathit{mkcell}\langle v_0, k_0 \rangle$ dans la solution entraîne la création de trois nouveaux canaux *get,set,s*. Le canal s contient toujours un message en attente, initialisé par v_0. Le nom de ce canal n'est pas communiqué : seuls les canaux *get* et *set* sont retournés via le canal k_0.

Lorsque le canal *get* reçoit un message $\mathit{get}\langle k \rangle$, la valeur en attente dans s est retournée via k, et recopiée dans s.

Lorsque le canal *set* reçoit un message $set\langle u, k\rangle$, la valeur en attente dans s est supprimée, remplacée par u, et un signal de retour est émis sur k.

2.2.6 Complément pour les migrations

L'arbre de locations

Avant d'aborder la question de la migration, il faut introduire la distribution dans le modèle. Pour cela, la machine chimique devient un ensemble de solutions, organisées en arborescence. Nous parlerons dorénavant de *location* pour désigner une solution dans le modèle distribué.

Chaque location est désignée par un nom, qui peut être transmis et reçu dans les messages du join-calcul, et sa position est donnée par une chaîne de noms de locations : $\vdash_\phi$ est une sous-location de $\vdash_\psi$ si ψ est préfixe de ϕ. L'arbre est correctement formé si pour un nom de location a il existe exactement une location ϕa. Par exemple $(\vdash_{aa})$ et $(\vdash_{ab} \parallel \vdash_{ba})$ ne sont pas corrects.

Quelques ajouts doivent alors être faits au join-calcul :

- un message se déplace vers la location qui abrite le canal sur lequel il a été émis
- une location peut créer une sous-location
- une location peut se déplacer sous une autre location
- une location peut s'interrompre (ou simuler une panne)
- une location peut détecter qu'une autre location est en panne

Complément à la syntaxe

$$P \stackrel{\text{def}}{=} \ldots \mid to\langle a, k\rangle \mid halt\langle\rangle \mid fail\langle a, k\rangle$$
$$D \stackrel{\text{def}}{=} \ldots \mid a[D : P]$$

Typiquement :

- $to\langle a, k\rangle$ déplace la location
- $a[D : P]$ crée une sous-location

Règles complémentaires

$$\begin{array}{rrcl} \textbf{comm} & \vdash x\langle\tilde{v}\rangle \parallel D \vdash & \longrightarrow & \vdash \parallel D \vdash x\langle\tilde{v}\rangle \quad (x \in dv[D]) \\ \textbf{str-loc} & a[D:P]\vdash_\phi & \rightleftharpoons & \vdash_\phi \parallel \{D\}\vdash_{\phi a}\{P\} \\ \textbf{move} & \vdash_{\psi a} \parallel \vdash_{\phi b} to\langle a,k\rangle & \longrightarrow & \vdash_{\psi a} \parallel \vdash_{\psi ab} k\langle\rangle \end{array}$$

2.2.7 Exemple

L'exemple suivant implémente le principe du code mobile
load_applet$\langle a\rangle \triangleright$
def b[applet$\langle y\rangle \triangleright$ **reply** computation$\langle y\rangle$ *to* applet
: to$\langle a\rangle$;**reply** applet *to* load_applet] **in** 0 $\vdash_s$
$\parallel \vdash_c$ **let** g = load_applet$\langle c\rangle$ **in** g$\langle 3\rangle$

En effet, le message load_applet$\langle c\rangle$ crée une sous-location b dont la première tâche consiste à se déplacer sous c.

2.2.8 Join-calcul et capacités

La connaissance du nom d'un canal du join-calcul est la condition nécessaire et suffisante pour avoir le droit de l'utiliser. Cette connaissance n'est jamais implicite : la règle **str-def** crée de nouveaux noms de canaux, qui ne sont pas devinables, et qu'il faut donc communiquer aux locations qui ont besoin d'y accéder.

Le nom d'un canal constitue donc une *capacité*. On peut considérer que le système de capacités est inscrit en dur dans le join-calcul :

- il est possible de donner ses capacités à autrui, puisqu'on peut communiquer les noms de canaux
- il est impossible de falsifier une capacité, puisqu'il est impossible de deviner le nom d'une capacité
- il est impossible de reprendre une capacité à quelqu'un, puisqu'il est impossible de retirer un message d'une location distante.

2.2.9 Implémentation du join-calcul et sécurité

Un point important du join-calcul est l'impossibilité d'utiliser un canal sans connaître son nom, et l'impossibilité de le deviner. Un nom de canal

ne peut être connu que si on le reçoit d'une location qui le reçoit d'une location ... qui le reçoit de la location qui a défini le canal. Chaque communication se faisant à l'initiative de celui qui donne l'information. Il n'y a pas en join-calcul la possibilité de contraindre une location à communiquer les noms de canaux qu'elle connaît.

Cette propriété est difficile à garantir au niveau de l'implémentation. En effet, on peut imaginer plusieurs moyens de subtiliser le nom d'un canal :

- installer un espion sur la ligne internet, et repérer dans les messages transmis les noms des canaux,
- analyser le contenu d'un agent immigré et le dépouiller de ses capacités : l'hôte peut en effet geler le fonctionnement d'un agent, et l'ausculter avec des outils qui n'ont rien à voir avec le join-calcul.

Ce dernier point semble imparable.

Pour continuer cependant à réfléchir aux problèmes de sécurité, nous sommes amenés à poser quelques hypothèses relativement acceptables.

Nous pouvons dans un premier temps faire une hypothèse relative à l'implémentation du join-calcul. Et considérer que celle-ci est correcte sur tous les sites.

En effet, nous pouvons considérer que le join-calcul est implémenté sous la forme d'un interpréteur et que celui-ci fait partie intégrante du système. Sous unix, quel utilisateur, lorsqu'il fait 'rlogin' se soucie de l'intégrité de l'implémentation du système unix auquel il accède ? Si on accepte l'idée que le join-calcul se présente sous la forme d'un interpréteur, on sécurise fortement l'importation d'agents, ceux-ci étant contraints de s'exécuter dans un cadre précis. Si on accepte l'idée que l'interpréteur appartient au système, on sécurise fortement l'exportation d'agents, ceux-ci ne pouvant se faire dépouiller de leurs capacités.

Le second point est celui des communications entre les différents agents. Si un agent souhaite communiquer avec un autre agent sur le même site, son message passe par l'interpréteur qui le retourne au destinataire. Puisque l'interpréteur appartient au système, la communication ne peut être interceptée par un tiers. Si maintenant le destinataire se trouve sur une autre

machine, le message émis passe par l'interpréteur du site émetteur qui le transmet à l'interpréteur du site destinataire, celui-ci le transmettant à l'agent destinataire. La faiblesse se trouve entre les deux interpréteurs, c'est-à-dire sur le réseau. Une solution, un peu lourde, existe cependant : il suffit d'attribuer à chaque interpréteur, c'est-à-dire à chaque machine une clef publique, utilisée pour ces échanges.

Une fois ces hypothèses faites, on peut considérer que l'implémentation du join-calcul est correcte et non altérable. Il serait faux de considérer que la question globale de la sécurité est résolue par ces hypothèses : au contraire, tout reste à faire, mais cette fois dans un cadre théorique clair.

Chapter 3

Join-calcul et cryptographie

3.1 Quelques propositions de base

Lorsqu'on parle de sécurité, on pense rapidement à la cryptographie, et notamment aux systèmes de clefs secrêtes, de clefs publiques et de clefs privées. Comment appliquer ces concepts au join-calcul ?

Le problème que nous allons étudier est le suivant : Bob veut envoyer un message M à Alice. De plus Bob veut être sûr qu'Alice l'a reçu, et Alice veut être sûre que Bob est bien l'émetteur.

Il y a plusieurs façons de coder ceci en join-calcul.

3.1.1 Le codage naïf de la cryptographie en join-calcul

La méthode la plus directe consiste à reproduire les systèmes de cryptographie à clef publique. Considèrons que Alice et Bob sont représentés chacun par une location, que nous appellerons pour simplifier les locations Alice et Bob.

Chacune de ces locations dispose de deux canaux qui prennent en entrée un entier et un nom de canal de retour, et retournent sur ce canal le résultat d'un calcul de type cryptographique ($x^p mod n$):

- un canal V calcule la fonction publique et son nom est communiqué à qui le souhaite
- un canal S calculant la fonction secrête dont le nom n'est jamais communiqué à l'extérieur

Toutefois cette méthode se contente d'implémenter l'existant en join-calcul, sans tirer partie des spécificités de ce calcul.

3.1.2 Le canal et la clef publique

Le rôle premier des systèmes à clef publique est de garantir l'émetteur que seul le récepteur supposé saura lire les messages cryptés avec sa clef publique.

Le canal du join-calcul joue un rôle similaire : par la règle **comm**, l'émetteur d'un message $x\langle\tilde{v}\rangle$ est assuré que celui-ci ne sera exploité que par la location qui a défini le canal x. Le canal joue donc le rôle d'une clef publique.

De plus, on remarque un invariant au cours du calcul d'une machine join-calcul distribuée : si à l'instant t, une location A abrite deux canaux u et v, alors à tout moment $t' > t$, la location A continue d'abriter les deux canaux u et v.

Une location peut être ainsi identifiée par un canal qu'elle abrite. On peut par conséquent supposer que toute location A définit un canal K_a dont elle diffuse publiquement le nom, et qui constitue de fait sa clef publique. Quelles que soient les transformations appliquées à l'arbre des locations, le fait d'envoyer un message sur K_a garantit que le message arrive en A, et n'est exploitable que par cette location.

3.1.3 Le canal et la clef secrête

La clef secrête se modélise également en join-calcul, par les canaux. Bob et Alice veulent communiquer directement, sans passer par le canal public. Pour se faire, Bob et Alice définissent chacun un nouveau canal, grace à la règle **str-def**, et se le communiquent, via un protocol à étudier. Dès lors, le canal créé par Bob est utilisé par Alice et par Alice seule, tant que Bob

ou Alice ne le communiquent pas à un tiers.

3.2 Adaptation de la logique des protocoles

L'*authentification* est l'action de déterminer l'identité d'un *principal* (une personne ou une machine). Ceci se fait typiquement par un échange de messages entre deux principaux A et B, où B souhaite s'authentifier auprès de A, en lui prouvant qu'il connaît certains secrets.

La description de ces messages échangés, ainsi que leur ordre forment un *protocole d'authentification*. Une question importante est la suivante : à la vue des messages échangés, que peuvent déduire les principaux ? Que prouvent ces messages ?

Pour répondre à ces questions, une logique a été développée, par Burrows-Abadi-Needham. Nous en proposons ici une adaptation au join-calcul, dans le sens précisé précédemment, c'est-à-dire que tout tourne autour des canaux, considérés commes des clefs.

3.2.1 Syntaxe

Quelques notations :

- P est un principal
- K est une clef, c'est-à-dire un canal
- T est un terme primitif
- M est un message
- F est une formule

$$
\begin{array}{rcl}
M & ::== & M, M \\
 & | & T \\
 & | & F \\
\\
F & ::== & P \models F \\
 & | & P \models\!\!> F \\
 & | & P \lhd \langle M \rangle \\
 & | & P \ \textbf{said} \ \langle M \rangle \\
 & | & P \ \textbf{says} \ \langle M \rangle \\
 & | & P \overset{M}{\rightleftharpoons} Q \\
 & | & \overset{K}{\longmapsto} P \\
 & | & F, F \\
 & | & \#\langle M \rangle \\
 & | & K\langle M \rangle
\end{array}
$$

Décrivons les différents symboles :

- $P \models F$: P croit que F est vrai, et agit comme tel,
- $P \models\!\!> F$: P a autorité sur F, les autres principaux lui font confiance sur la véracité de F,
- $P \lhd \langle M \rangle$: P reçoit le message M, on suppose que P est capable de savoir si c'est lui-même qui a envoyé M ou non.
- P **said** $\langle M \rangle$: P a envoyé le message M, mais on ne sait pas exactement quand,
- P **says** $\langle M \rangle$: P a envoyé le message M au cours du protocole actuel,
- $P \overset{M}{\rightleftharpoons} Q$: M est un secret partagé par P et Q, et éventuellement par des principaux auxquels P ou Q font confiance,
- $\#\langle M \rangle$: le message M est récent, c'est-à-dire qu'il n'a pu être construit que depuis le début du protocole courant,
- $K\langle M \rangle$: le message M est envoyé sur le canal K.

La notion de temps est importante dans la logique, mais elle est simple : on distingue deux époques, le présent et le passé. Le présent correspond au protocole d'authentification en cours.

3.2.2 Règles d'inférence

Les règles d'inférence de la logique sont les suivantes :

Quelques règles techniques

$$\frac{P \models F \,,\, \frac{F}{G}}{P \models G} \quad , \quad \frac{P \models F \,,\, P \models G}{P \models F,G}$$

$$\frac{P \models F}{P \models P \models F} \quad , \quad \frac{P \lhd \langle X \rangle}{P \models P \lhd \langle X \rangle} \quad , \quad \frac{\overset{K}{\longmapsto} P}{P \models \overset{K}{\longmapsto} P}$$

$$\frac{\#\langle X \rangle}{\#\langle X,Y \rangle}$$

$$\frac{P \lhd \langle X,Y \rangle}{P \lhd \langle X \rangle} \quad , \quad \frac{P \textbf{ said } \langle X,Y \rangle}{P \textbf{ said } \langle X \rangle} \quad , \quad \frac{P \textbf{ says } \langle X,Y \rangle}{P \textbf{ says } \langle X \rangle}$$

$$\frac{P \models F_1,F_2}{P \models F_1}$$

Règle de visualisation

$$\frac{K\langle M \rangle \,,\, \overset{K}{\longmapsto} P}{P \lhd \langle M \,,\, \overset{K}{\longmapsto} P \rangle}$$

Seul le possesseur d'un canal peut voir les messages qui sont émis dessus.

Règle d'authentification

$$\frac{P \overset{Y}{\rightleftharpoons} Q \,,\, R \lhd \langle X,Y \rangle}{Q \textbf{ said } \langle X,Y \rangle}$$

En supposant que P n'est pas l'auteur du message $\langle X, Y \rangle$, la présence d'un secret partagé dans un message permet de déduire l'identité de l'émetteur.

Règle de fraîcheur

$$\frac{\#\langle X \rangle \, , \, P \ \textbf{said} \ \langle X \rangle}{P \ \textbf{says} \ \langle X \rangle}$$

La présence dans un message d'un élément appartenant au présent permet de situer son émission dans le présent.

Règle de juridiction

$$\frac{P \ \models\!\Rightarrow X \, , \, P \ \textbf{says} \ \langle X \rangle}{X}$$

Si un message appartient au présent, et provient d'un principal qui a autorité sur lui, il est considéré comme vrai.

3.2.3 Mise en œuvre

Un protocole est typiquement défini par la liste des messages échangés, ainsi que par leur format. La première étape de mise en œuvre de la logique consiste à écrire dans le formalisme précédent les messages échangés : on y trouve d'une part les informations écrites en clair dans le protocole, mais également les déductions sous-jacentes ; en effet la réponse à un message est parfois conditionnée au fait que ce message a convaincu d'un certain point.

Prenons l'exemple décrit précédemment : A veut envoyer un message M à B, de telle sorte que ce soit bien B qui le reçoive, et que B sache que A en est l'auteur. On suppose que A et B disposent chacun d'un canal public K_A et K_B. Le protocole est le suivant :

- $A \to B$: envoie le message (A, M) sur le canal K_B,
- $B \to A$: retourne le message (M, r) sur le canal K_A, où r est un nouveau canal de B,

- $A \rightarrow B$: si A reconnaît être l'émetteur de M, il répond sur r.

Ceci se traduit dans la logique par :

1. $K_B\langle A, M\rangle$
2. $K_A\langle M \ , \ B \lhd \langle M\rangle \ , \ \overset{r}{\longmapsto} B\rangle$
3. $r\langle A \ \textbf{said} \ \langle M\rangle\rangle$

On fait alors les hypothèses suivantes :

- $\overset{K_A}{\longmapsto} A \ , \ \overset{K_B}{\longmapsto} B \ , \ \overset{r}{\longmapsto} B$
- $A \ \models A \overset{M}{\rightleftharpoons} B \ , \ A \ \models \#\langle M\rangle$
- $B \ \models A \overset{\overset{r}{\longmapsto} B}{\rightleftharpoons} B \ , \ B \ \models \#\langle \overset{r}{\longmapsto} B\rangle$
- $A \ \models B \ \models\!\!\Rightarrow \overset{r}{\longmapsto} B \ , \ A \ \models B \ \models\!\!\Rightarrow B \lhd \langle M\rangle$
- $B \ \models A \ \models\!\!\Rightarrow A \ \textbf{said} \ \langle M\rangle$

A partir de là, on applique les règles d'inférences :

1. $K_B\langle A, M\rangle$
- $B \lhd \langle A, M, \overset{K_B}{\longmapsto} B\rangle$
- $\mathbf{B} \lhd \langle \mathbf{M}\rangle$
2. $K_A\langle M \ , \ B \lhd \langle M\rangle \ , \ \overset{r}{\longmapsto} B\rangle$
- $A \lhd \langle M \ , \ B \lhd \langle M\rangle \ , \ \overset{r}{\longmapsto} B, \ \overset{K_A}{\longmapsto} A\rangle$
- $A \ \models B \ \textbf{said} \ \langle M \ , \ B \lhd \langle M\rangle \ , \ \overset{r}{\longmapsto} B, \ \overset{K_A}{\longmapsto} A\rangle$
- $A \ \models B \ \textbf{says} \ \langle M \ , \ B \lhd \langle M\rangle \ , \ \overset{r}{\longmapsto} B, \ \overset{K_A}{\longmapsto} A\rangle$
- $\mathbf{A} \ \models \mathbf{B} \lhd \langle \mathbf{M}\rangle \ , \ \mathbf{A} \ \models\!\!\Rightarrow \overset{\mathbf{r}}{\longmapsto} \mathbf{B}$
3. $r\langle A \ \textbf{said} \ \langle M\rangle\rangle$
- $B \lhd \langle A \ \textbf{said} \ \langle M\rangle, \overset{r}{\longmapsto} B\rangle$

- $B \models A$ **said** $\langle A$ **said** $\langle M \rangle, \overset{r}{\longmapsto} B \rangle$
- $B \models A$ **says** $\langle A$ **said** $\langle M \rangle, \overset{r}{\longmapsto} B \rangle$
- $\mathbf{B \models A}$ **said** $\langle \mathbf{M} \rangle$

Un des grands intérêts de cette logique est de mettre en évidence les hypothèses faites parfois implicitement. Ainsi dans l'exemple précédent doit on faire l'hypothèse que le message M est récent. On fait également l'hypothèse (la dernière) que A doit être cru lorsqu'il dit qu'il est l'auteur de M. Cela fait intervenir une certaine bonne foi des deux parties. C'est, on l'aura compris, un protocole simplifié, qui permet de montrer une application de la logique.

3.3 Implémentation en join-calcul

Comme nous allons le voir, l'implémentation de l'exemple simple précédent se révèle délicate en join-calcul.

Une implémentation *premier-jet* pourraît être (on ne détaille pas dans cet exemple le mécanisme qui permet de passer de l'identité A à la clef publique K_A):

Sur A:
$K_A\langle tag,r\rangle \rhd tag\langle r\rangle$
$Send\langle B, M\rangle \rhd \mathbf{def}\ \ tag\langle r\rangle \rhd r\langle\rangle\ \ \mathbf{in}\ \ K_B\langle A, M, tag\rangle$

Sur B:
$K_B\langle A, M, tag\rangle \rhd \mathbf{def}\ \ r\langle\rangle \rhd \mathrm{OK}\langle A, M\rangle\ \ \mathbf{in}\ \ K_A\langle tag, r\rangle$

Cette implémentation est fausse : B pourraît se faire passer pour A auprès d'un tiers C de deux manière différentes. Première manière : il suffit que B émette vers C le message $K_C\langle A, M', tag\rangle$, et C croira que le message M' provient de A. On suppose que C fonctionne comme B le fait plus haut, et que B est modifié :

Sur A:
$K_A\langle tag,r\rangle \rhd tag\langle r\rangle$
$Send\langle B, M\rangle \rhd \mathbf{def}\ \ tag\langle r\rangle \rhd r\langle\rangle\ \ \mathbf{in}\ \ K_B\langle A, M, tag\rangle$

Sur B:
$K_B\langle A, M, tag\rangle \rhd K_C\langle A, M', tag\rangle$

Sur C:
$K_C\langle A, M, tag\rangle \rhd \textbf{def}\ \ r\langle\rangle \rhd \mathrm{OK}\langle A, M\rangle\ \ \textbf{in}\ \ K_A\langle tag, r\rangle$

Deuxième manière, C peut se faire passer pour A auprès de B en utilisant à son profit le canal K_A qui n'a aucune garantie que le message $tag\langle r\rangle$ ne va pas migrer : tag n'est pas nécessairement local. Cette méthode est plus dangereuse, car elle peut se produire n'importe quand, et pas uniquement lorsque A envoie un message. Les codes de A et de B ne sont pas modifiés, le code de C ressemble au code de A.

Sur A:
$K_A\langle tag,r\rangle \rhd tag\langle r\rangle$
$Send\langle B, M\rangle \rhd \textbf{def}\ \ tag\langle r\rangle \rhd r\langle\rangle\ \ \textbf{in}\ \ K_B\langle A, M, tag\rangle$

Sur B:
$K_B\langle A, M, tag\rangle \rhd \textbf{def}\ \ r\langle\rangle \rhd \mathrm{OK}\langle A, M\rangle\ \ \textbf{in}\ \ K_A\langle tag, r\rangle$

Sur C:
$Send\langle B, M\rangle \rhd \textbf{def}\ \ tag\langle r\rangle \rhd r\langle\rangle\ \ \textbf{in}\ \ K_B\langle A, M, tag\rangle$

La première méthode peut être fixée, en autorisant la comparaison de noms d'utilisateurs. Le code devient:

Sur A:
$K_A\langle B, tag,r\rangle \rhd tag\langle B, r\rangle$
$Send\langle B, M\rangle \rhd \textbf{def}\ \ tag\langle X, r\rangle \rhd \textbf{if}\ \ X = B\ \ \textbf{then}\ \ r\langle\rangle\ \ \textbf{in}\ \ K_B\langle A, M, tag\rangle$

Sur B:
$K_B\langle A, M, tag\rangle \rhd \textbf{def}\ \ r\langle\rangle \rhd \mathrm{OK}\langle A, M\rangle\ \ \textbf{in}\ \ K_A\langle B, tag, r\rangle$

La deuxième méthode est plus difficile à fixer. Il faut d'une manière ou d'une autre que A puisse vérifier qu'il est bien l'auteur du message, c'est-à-dire que le tag qu'il reçoit lui appartient. Une solution est de définir un tableau :

- par le canal *store*, on y stocke des triplets (canal, identité, message)
- par le canal *get*, on y récupère le canal associé à un couple (identité, message)

Une solution est alors :

Sur A:
$K_A\langle M, r, B\rangle \triangleright$ **let** $resp = get\langle B, M\rangle$ **in** $resp\langle r\rangle$
$Send\langle B, M\rangle \triangleright$ **def** $respond\langle r\rangle \triangleright r\langle\rangle$ **in** store$\langle respond, B, M\rangle$; $K_B\langle A, M\rangle$

Sur B:
$K_B\langle A, M\rangle \triangleright$ **def** $r\langle\rangle \triangleright \mathrm{OK}\langle A, M\rangle$ **in** $K_A\langle M, r, B\rangle$

Une autre approche consiste à définir des canaux locaux, c'est-à-dire qui ne possèdent pas la règle **comm**. Ceci aboutit à la proposition de typage suivante :

Type	**Notation**	**Remarques**
entier	$\#u$	comparaison possible
canal local	$@u$	pas de règle comm
canal global	u	

La solution s'écrit alors :

Sur A:
$K_A\langle \#X, @tag, r\rangle \triangleright tag\langle X, r\rangle$
$Send\langle \#B, \#M\rangle \triangleright$ **def** $@tag\langle \#X, r\rangle \triangleright$ **if** $X = B$ **then** $r\langle\rangle$ **in** $K_B\langle A, M, tag\rangle$

Sur B:
$K_B\langle \#A, \#M, @tag\rangle \triangleright$ **def** $r\langle\rangle \triangleright \mathrm{OK}\langle A, M\rangle$ **in** $K_A\langle B, tag, r\rangle$

3.4 Observations

Nous avons montré dans ce chapitre qu'il est possible, mais compliqué, de réaliser une fonction qui n'est pas dans le join-calcul de base, et qui consiste à pouvoir s'assurer de l'origine d'un message. Non seulement l'implémentation nécessaire est relativement lourde, mais encore avons-nous ignoré un problème pourtant conséquent, celui de la notion d'utilisateur. En effet, nous avons implicitement assimilé la notion d'utilisateur à celle de location. De plus, en cherchant une manière d'invariant aux migrations, nous avons considéré que les caractéristiques extérieures des locations –au premier rang desquelles la confiance accordée par les tiers– étaient insensibles aux déplacements.

Chaque location est ainsi assimilée à un utilisateur, c'est-à-dire à un canal qui sert de clef publique. Même si en ajoutant des mécanismes assez simples, il est possible de regrouper plusieurs locations sous une même clef publique –ce qui centralise cependant les demandes d'authentification–, un tel système, qui nie la notion d'utilisateur humain, et accessoirement de responsabilité, aboutit à une multiplication des identités inconnues, et oblige ou bien à se méfier de tout le monde, puisque tout le monde est inconnu, ou bien à introduire des notions de transitivité de confiance, notions bien discutables.

Quoiqu'intéressante à étudier, l'approche protocolaire montre ses limites.

Chapter 4

L'utilisateur retrouvé

La question importante pour les processus mobiles n'est peut-être pas de savoir à quoi ils servent, mais plutôt à qui. Par un effort d'imagination et d'anticipation, on peut imaginer une situation où le processus mobile s'est banalisé et est devenu le standard de l'informatique en réseau. Dans une telle situation, on envisage mal d'accueillir un agent, sans savoir ce qu'il fait, ni ce qu'il est sensé faire, ni pour qui il travaille. Même si la sécurité est totale, on hésitera à fournir du temps machine à une application inconnue, pour la raison même qui est à la base de la mobilité des processus, les performances.

4.1 Des capacités aux listes d'accès

Les capacités et les listes d'accès sont deux façons différentes de représenter une même chose. Il s'agit dans les deux cas d'établir un lien entre des utilisateurs et des ressources. On peut représenter les choses sous la forme d'un tableau : chaque colonne correspond à une ressource R, et chaque ligne à un utilisateur U. Dans chaque case du tableau (R, U), on indique les droits d'accès de U sur R. Alors un système à listes d'accès considère que ce tableau est un ensemble de colonnes, alors qu'un système à base de capacités considère que le tableau est un ensemble de lignes.

En effet, dans un système à liste d'accès, chaque ressource gère la liste des identités des utilisateurs autorisés à accéder à la ressource. Tout utilisateur doit donc s'identifier pour pouvoir être accepté. Sous Unix, la gestion des droits d'accès aux fichiers est un système à liste d'accès simplifiée : cette

liste reconnaît trois *ensembles* : le propriétaire, un groupe d'utilisateurs, et tous les autres. Pour chaque ensemble, la liste d'accès détermine trois droits : lecture, écriture, exécution.

En revanche, dans un système à capacités, la ressource émet des tickets qui sont conservés par les utilisateurs. Ceux-ci doivent les représenter aux ressources pour pouvoir accéder à celles-ci. La ressource doit authentifier ces tickets : cette authentification ne fait pas intervenir l'identité de l'utilisateur émetteur, mais plutôt des procédés cryptographiques de signature.

Un système exclusivement fondé sur les capacités garde un pouvoir expressif fort : on peut par exemple créer une protection de fichiers de type unix. Le mot de passe est en fait une capacité, qui permet d'accéder auprès d'un serveur d'authentification à l'ensemble des capacités détenues par un utilisateur. Ces capacités donnent des droits d'accès aux fichiers détenus par l'utilisateur, ainsi qu'aux fichiers accessibles par les groupes auxquels appartient l'utilisateur.

La difficulté consiste à gérer les fonctions de maintenance des droits d'accès : typiquement, pour réaliser un 'chmod' restreignant les droits d'accès, le propriétaire d'un fichier doit invalider les capacités vers ce fichier. Ceci est facile, mais un peu sale, car il est difficile dans un système distribué de prévenir tout le monde qu'une capacité a été invalidée : le système conserve donc de vieilles capacités inutiles. Pour effacer un utilisateur d'un groupe, il faut recréer toutes les capacités du groupe et les redonner uniquement aux autres membres du groupe.

Un système fondé sur les capacités est, de plus, fragile : la fuite d'une seule des capacités d'un utilisateur constitue un trou de sécurité, alors qu'avec un système à liste d'accès, en supposant qu'il est correctement implémenté, le seul trou réside dans la fuite du mot de passe permettant l'identification de l'utilisateur. Il est intuitivement plus facile de conserver un secret, que d'en conserver beaucoup. En effet, un objectif important de la sécurité, au-delà de protéger un utilisateur d'applications malveillantes, est de protéger l'utilisateur des erreurs de programmation : il faut pouvoir l'assurer que ses capacités ne pourront pas se diffuser, même si son code est mal écrit et tente de communiquer une capacité.

Pour illustrer ceci en Unix, si un utilisateur A écrit un programme utilisant malencontreusement un fichier f et le donne à un utilisateur B, B ne pourra l'utiliser que s'il a lui-aussi accès à f. En réglant correctement

les drois d'accès à f, l'utilisateur A peut donc se prévenir de ses propres erreurs de programmation.

Enfin, est-il possible de se passer de la notion d'identité ? En effet, lorsqu'un utilisateur demande l'accès à une ressource, sur quels critères cet accès est-il accordé ou refusé ? Si ce n'est pas sur l'identité de l'utilisateur que ce choix se fait, ce ne peut être que par la présentation d'une capacité, une capacité de demande de capacité. Pour les raisons évoquées précédement, un tel système est fragile. Il est important de pouvoir procéder à des contrôles réguliers.

4.2 Notion d'utilisateur et join-calcul

Introduire une notion d'utilisateur dans le join-calcul est un passage obligé vers l'utilisation de listes d'accès. Plusieurs questions se posent :

- Quelle est la nature d'une identité ?
- A quel moment l'identité intervient-elle dans le join-calcul ?
- Quel est le rapport entre identité et location ?

4.2.1 Nature d'une identité

Nous définirons deux termes : identifiant et identité.

Identifiant

Un identifiant est un nom (une chaîne de caractères) désignant un individu ou un groupe. C'est par exemple le nom d'un utilisateur, ou le nom d'un projet. On suppose qu'un identifiant est unique : un même identifiant ne peut désigner deux choses distinctes.

Identité

Une identité est un ensemble d'identifiants, éventuellement réduit à un singleton. Une identité permet de caractériser complètement un utilisateur : on y trouve son nom, ainsi que tous les groupes dont il est membre.

4.2.2 Identité et join-calcul

L'introduction d'une notion d'identité a pour rôle de limiter les possibilités d'un programme écrit en join-calcul. Cette fonction de filtre peut intervenir de diverses manières : empêcher le déplacement d'un message en contrôlant l'accès à un canal, ou restreindre les possibilités de migration.

4.2.3 Identité et locations

Que le filtrage concerne les messages ou les migrations, l'identité est attachée aux locations : dans le premier cas, la location est considérée comme responsable des messages qu'elle émet. Son identité est donc attachée aux messages qu'elle envoie. Dans le second cas, c'est directement l'identité de la location migrante, ainsi éventuellement que celle des locations avec lesquelles elle communique qui est prise en compte pour décider d'accepter ou de refuser la migration.

Trois points sont alors à préciser :

- qu'elle est l'identité d'une location créée par la règle **str-loc** ?
- comment évolue l'identité d'une location au cours d'une migration par la règle **move** ?
- comment introduit-on une nouvelle identité dans le système ?

4.3 Approche de type Unix

4.3.1 Rappel

Rappellons brièvement comment les accès aux fichiers sont régis sous Unix.

Il y a deux types d'identifiants : les numéros d'utilisateurs, et les numéros de groupes. On distingue un utilisateur particulier unique, appelé *root*, qui dispose de certains droits supplémentaires.

Chaque fichier conserve le numéro de son propriétaire et le numéro du groupe auquel celui-ci appartient, et dispose d'un jeu de droits d'accès pour le *propriétaire*, le *groupe propriétaire* et le *reste du monde*.

Chaque processus se voit attribuer un numéro d'utilisateur réel et un numéro d'utilisateur effectif. L'utilisateur réel est typiquement celui qui

a lancé le programme. L'utilisateur effectif est soit celui qui a lancé le programme, soit le propriétaire du programme. C'est avec les droits de l'utilisateur effectif que le programme sera exécuté. En effet, on distingue deux sortes de programmes :

- les programmes *setuid*, dont l'utilisateur effectif est en fait le propriétaire du programme
- les programmes *non-setuid*, dont l'utilisateur effectif est l'utilisateur réel.

Le choix entre programme *setuid* et programme *non-setuid* est fait par le propriétaire du programme.

Les processus reçoivent également des numéros de groupe réel et de groupe effectif, gérés de la même manière.

Un processus ne peut changer d'identité en cours d'exécution, sauf s'il appartient à *root*. Dans ce cas seulement, un processus peut revêtir toute autre identité.

Lors de l'accès à un fichier, le numéro d'utilisateur effectif du processus effectuant l'accès est comparé au numéro de propriétaire du fichier. En cas d'égalité, ce sont les droits *propriétaire* du fichier qui sont pris en compte. Puis on compare les numéros du groupe effectif du processus et du groupe propriétaire du fichier. En cas d'égalité, ce sont les droits *groupe propriétaire* du fichier qui sont pris en compte, sinon ce sont les droits *reste du monde*.

4.3.2 Gestion des identités des locations

Nous effectuons une transposition du système de type Unix au join-calcul, en faisant l'analogie entre d'une part *processus* et *location*, et d'autre part *fichier* et *canal*.

Attributs des locations

A chaque location, nous attribuons :

- une identité réelle,

- une identité effective,
- un flag indiquant si la location est *setuid* ou *non-setuid*.

Ces informations sont gérées par le système. On conserve une notion de *root*, qui correspond à un super-utilisateur pour un ensemble d'utilisateurs. Plusieurs *root* peuvent donc coexister.

Règles de propagation

Les règles de propagation détermine comment les identités réelles et effectives se transmettent lors des créations et des migrations des locations.

- $\vdash_b a[D : P]$: lors de la création par b d'une sous-location a, les attributs identités réelles et effectives de b sont transmises intégralement à la nouvelle location, le flag *setuid* de a est choisi par b (il convient pour cela d'aménager la syntaxe),
- $to\langle a\rangle$: lors d'une migration, si la location migrante est de type *non-setuid*, l'identité effective de cette location et, récursivement de toutes ses sous-locations *non-setuid* est réglée à l'identité effective de la location d'accueil ; sinon, si la location est de type *setuid*, aucune modification d'identité n'est effectuée,
- une location *root* peut créer une sous-location avec l'identité d'un des membres du groupe dont ce *root* est super-utilisateur.

Remarques

On constate une fragilité certaine dans cette méthode : elle concerne les migrations. En effet, dans le join-calcul tel qu'il se présente aujourd'hui, la seule règle de migration est la règle **move**. Or cette règle peut être invoquée par toute location connaissant le nom d'une autre location, appelée location d'accueil, sans que celle-ci puisse refuser la migration ni même en être avertie.

Le contrôle des migrations peut être vu sous forme capacitive : le nom d'une location constitue une capacité pour migrer sous cette location. On peut invalider une location en y effectuant un $halt\langle\rangle$. Pour pouvoir donner une autorisation d'accueil et garder la possibilité de la reprendre, une location a doit donc procéder de manière subtile ; au prélable, on a les principes suivants :

- a ne communique jamais son nom de location
- a crée une sous-location tampon b que a peut arrêter à tout moment : a conserve un canal vers b qui commande un $halt\langle\rangle$.
- a communique b aux locations qu'il souhaite voir venir : celles-ci deviennent sous-locations de b et non de a.

Supposons à présent qu'il y a sous b un immigrant indésirable c. Pour éliminer un immigrant indésirable :

- a crée une sous-location tampon b', comparable à b
- a communique b' aux immigrants placés sous b à l'exception de c
- a attend que ceux-ci aient migré sous b'
- a détruit b et donc c

Ceci n'est cependant envisageable que si une location dispose de moyens de détection de ses sous-locations.

Nous avons vu qu'une location *non-setuid* prenait l'identité effective de son hôte lorsqu'elle migrait. Le fait qu'une location ne puisse contrôler les immigrations constitue un trou grave. En effet, il suffit à une location de type *non-setuid* de connaître une location A pour pouvoir se déplacer sous A et agir au nom de l'identité de A.

Dans l'optique inverse, où la migration se fait par un $get\langle a\rangle$, le problème reste entier, car alors une location n'est plus maîtresse d'elle-même, et peut donc changer d'identité effective sans son accord.

Il semble raisonnable de considérer qu'une migration doit être négociée entre le migrant et l'hôte, et que les deux partis sachent quand et si la migration a eu lieu.

4.3.3 Rôle des identités dans les communications

Nous disposons à présent d'un système qui définit complètement les identités des locations. Il faut définir à présent comment elles interviennent dans le calcul.

Définition des canaux

Lorsqu'un canal est créé sur une location par la règle **str-def**, un ensemble d'identifiants lui est adjoint, que nous appelleront l'identité du canal. Cette identité est choisie par la location qui crée le canal. Cela suppose que les identifiants sont accessibles au niveau du join-calcul, et ne restent pas cachés dans le système.

Filtrage des messages

Lorsqu'un message $u\langle x\rangle$ est produit par une location, une identité lui est attachée : l'identité effective de la location émettrice. Grace à la règle **comm**, le message atteint la location qui a défini le canal u. On dispose alors de deux identités : l'identité du message $u\langle x\rangle$, et l'identité du canal u. Si les deux identités ont au moins un identifiant commun, le message est pris en compte. Si les deux identités n'ont aucun identifiant commun, le message est détruit.

4.3.4 Observations

Sous Unix, le rôle des programmes *setuid* est de donner à un utilisateur un accès restreint à une ressource qui ne lui appartient pas : cet accès ne peut se faire que par un programme inaltérable qui accède à la ressource d'une certaine manière. L'exemple le plus démonstratif est la commande *passwd* d'Unix qui permet à un utilisateur quelconque de modifier le fichier contenant les mots de passe de tous les utilisateurs d'une machine. L'utilisateur ne doit pouvoir modifier que son mot de passe dans ce fichier.

Un programme *setuid* permet donc d'affiner la notion de droits d'accès, le simple triplet de droits (lecture, écriture, exécution) étant trop grossier.

L'adaptation que nous proposons au join-calcul permet de conserver ces caractéristiques. Ce faisant, nous avons défini un lien implicite entre responsabilité et migration. En effet, l'identité effective d'une location ne peut changer qu'au cours d'une migration.

Une location *setuid* correspond à une location accueillie par l'hôte : elle ne change ni d'identité, ni de nature. La seule motivation de la migration est une question de performance : elle peut se rapprocher d'un de ses correspondants privilégiés.

Une location *non-setuid* correspond à une location donnée à l'hôte : elle en prend l'identité. La motivation de l'hôte est de prendre la responsabilité de l'immigrant, ce qui sous-entend un lien de confiance de l'hôte vers l'immigrant.

4.3.5 Rôle des escales

Cependant, une des caractéristiques des processus d'Unix n'est pas conservée dans le join-calcul : alors qu'un processus ne peut changer d'identité, une location *non-setuid* peut changer d'identité à chaque migration. Ceci pose quelques problèmes.

Considérons une location X *non-setuid* et trois locations A, B et C. On suppose que X est initialement une sous-location de A, et on examine deux scénari : dans le premier cas, la location X se déplace directement sous C, dans le second cas, la location X se déplace sous B puis sous C. La question qui se pose est : en quoi l'escale sous B peut-elle changer la confiance que C accorde à X, une fois que X se trouve sous C ?

Dans le cas de l'escale sous B, X prend momentanément l'identité de B, et obtient par conséquent une relation privilégiée avec B. Ce faisant, X prend connaissance de certaines capacités de B, et inversement.

Lorsque X migre ensuite vers C, son identité change, mais pas son contenu. En migrant, il emporte un lot de capacités pointant vers B, et B conserve les siennes vers X. Les capacités pointant vers B sont perdues si les filtres associés n'acceptent pas l'identité de C, mais pas celles de B vers X, ce qui peut gêner C.

L'escale n'est donc pas neutre.

4.3.6 Contrôle de la diffusion

La sécurité est en général un système permettant de contraindre la migration d'informations. On peut la voir au moins sous deux aspects différents :

- empêcher l'information de se diffuser
- empêcher l'utilisation de l'information diffusée.

La première approche semble plus délicate, car elle consiste à contrôler des échanges distants du propriétaire de l'information : si A donne une information x à B, comment empêcher B de la transmettre à C ?

La seconde approche laisse B diffuser l'information à C, mais, dans le cas où x est une capacité vers une ressource de A, empêche C de se servir de cette ressource. Cela correspond sous Unix à laisser un répertoire en lecture pour tous, en rendant inaccessibles tous les fichiers qu'il contient.

Les deux approches ne sont pas mutuellement exclusives.

Empêcher la diffusion *à la source* est une idée séduisante qui trouve sa limite dans les relais. Si A donne une capacité u à B, et si par un moyen ou un autre, on sait empêcher B de transmettre u à C, il est toujours possible à B de servir de relai à C, en lui donnant un canal v, et en transmettant sur u ce qu'il reçoit sur v.

Le relai n'est cependant pas transparent : les requêtes faites sur un canal le sont au nom du relai, et non de l'émetteur réel. Il y a donc là un risque de détection pour le relai.

Puisque les communications passent par le système gérant le join-calculus, il est possible d'ajouter des filtres sur le contenu du message et non pas sur le canal utilisé. A une information x donnée, on affecte un ensemble d'identifiants. Et le message ne peut être exploité que par une location possédant au moins un de ces identifiants dans son identité.

Chapter 5

Les deux mobilités

Les processus mobiles introduisent deux idées importantes en informatique. D'un côté, un processus peut se déplacer d'une machine à l'autre, de manière transparente, en conservant ses liens vers les autres processus. De l'autre côté, son utilité peut évoluer au cours de son exécution. Son caractère nomade lui imprime une sorte d'indépendance et de liberté : un processus naît au service d'un autre, puis peut travailler pour un troisième, et changer encore.

Lorsqu'un utilisateur souhaite utiliser un programme, il ne va pas, comme dans les systèmes classiques, le chercher sur un disque, l'installer en mémoire, avec ses droits ou ceux de son propriétaire ; en join-calcul, il demande au propriétaire de créer une copie du programme, et de la lui transmettre. Eventuellement, cette copie est également capable de se dédoubler pour qu'une nouvelle copie soit transmise à un autre utilisateur.

Le changement de propriété est donc à la base des processus mobiles. C'est une seconde forme de mobilité, que nous qualifierons de *mobilité de responsabilité*.

Jusqu'à présent, nous avons :

- soit nié la mobilité de responsabilité, dans le cas de l'approche *cryptographique*,
- soit lié mobilité physique et mobilité de responsabilité, dans le cas de l'approche de type Unix.

Dans ce chapitre, nous étudions les liens entre les deux mobilités, et proposons une solution.

5.1 Conditions de la mobilité

Dans cette partie, nous nous plaçons dans un cadre général, et tentons de définir quels intervenants sont concernés par une migration physique ou une migration de responsabilité, ainsi que les conditions que ceux-ci sont susceptibles de poser à de telles migrations.

5.1.1 Mobilité physique

Dans le cas d'une mobilité physique seule, les relations entre la location migrante et les autres locations ne sont pas modifiées. Les seuls éléments à prendre en compte sont les suivants :

- en se déplaçant, une location déplace la charge de travail qu'elle représente d'une machine vers une autre,
- en se déplaçant, une location modifie sa vulnérabilité aux pannes.

Les différents intervenants sont donc :

- la location migrante, qui est la première concernée par ses déplacements,
- la location d'accueil, qui va recevoir une charge affectant ses performances,
- de manière moins évidente, les locations possédant un lien vers la location migrante peuvent être sensibles au changement de vulnérabilité.

La mobilité physique est à l'initiative de l'agent migrant, qui, en se déplaçant, souhaite typiquement se rapprocher de certaines ressources. Les paramètres qui poussent un intervenant à accepter ou refuser une migration sont les suivants :

- le migrant considère la position physique de l'hôte,
- l'hôte s'inquiète de l'identité du migrant et de ce qu'il est sensé faire, pour évaluer s'il est prêt à lui céder une partie de ses ressources de calcul,

- les locations liées à la location migrante fondent leur décision sur la nature physique de l'hôte, pour évaluer sa fiabilité.

5.1.2 Mobilité de responsabilité

Une mobilité de responsabilité correspond simplement à un changement d'indentité d'une location. Le seul élément à prendre en compte est que, en changeant ainsi d'identité, une location gagne ou au contraire perd la confiance d'autres locations.

Les différents intervenants sont :

- le ou les anciens responsables
- le ou les nouveaux responsables
- toutes les locations qui possèdent un lien vers la location migrante. Par lien, on entend canal de communication, dans un sens ou dans l'autre.

La mobilité de responsabilité est à l'initiative du nouveau responsable, qui souhaite prendre le contrôle de la location. Les considérations à prendre en compte sont les suivantes :

- le nouveau responsable veut contrôler les liens de la location
- l'ancien responsable accepte de perdre sa responsabilité en fonction de l'identité du nouveau responsable
- les locations liées veulent savoir qui hérite des liens que la location migrante avait tendus vers elles.

5.1.3 Observations

Les deux mobilités sont bien différentes. Cependant on remarque une certaine similitude entre notamment les intervenants.

De plus un élément que nous n'avons pas évoqué jusqu'à présent et que nous n'allons pas développer rapproche les deux mobilités, il s'agit de l'implémentation. En effet, en cas d'implémentations non fiables, il est naturel d'accepter plus facilement la responsabilité d'une location locale, et donc d'attacher la mobilité de responsabilité à la mobilité physique.

5.2 Acceptation de la responsabilité

5.2.1 Vérification d'une location

Pour qu'une location accepte de prendre une autre location sous sa responsabilité (nous parlerons d'hôte et de migrant, même si la migration n'est pas ici nécessairement physique), il faut que l'hôte puisse effectuer des vérifications sur le migrant.

On peut imaginer plusieurs types de vérification :

- vérification du *code*,
- vérification de l'identité avant migration,
- vérification des liens.

La vérification du code n'est guère envisageable, car le lien entre le code et ce qu'il produit n'est pas trivial. La vérification de l'identité est simple, celle-ci étant fournie lors de la négociation entre le migrant et l'hôte.

La vérification des liens est une question qui présente quatre cas différents, que l'on peut regrouper par deux : liens IN et liens OUT. Nous considérerons le cas suivant : une location A s'apprête à migrer (du point de vue des responsabilités) sous B. Or A possède des liens vers une location X. Nous appellerons *lien* un nom de canal présent en tant que variable libre dans une location.

5.2.2 Liens *IN*

Un lien IN est dans notre exemple un nom de canal de A présent dans la location X ; c'est donc un lien qui permet à X d'envoyer des messages à A.

Deux cas se présentent :

- L'hôte B n'accepte pas A tant que X conserve un lien vers A,
- X ne veut pas conserver de lien vers A si A migre vers B.

On peut reformuler le second point de cette manière : X empêche A de migrer vers B tant qu'il conserve ce lien IN vers A.

5.2.3 Liens *OUT*

Un lien *OUT* est dans notre exemple un nom de canal de X présent dans la location A ; c'est donc un lien qui permet à A d'envoyer des messages à X.

Deux cas se présentent :

- L'hôte B n'accepte pas A tant que A conserve un lien vers X,
- X ne veut pas que A conserve ce lien vers lui si A migre vers B.

On peut reformuler le second point de cette manière : X empêche A de migrer vers B tant que A conserve un lien *OUT* vers X.

5.2.4 Négociations

La mobilité de responsabilité fait intervenir plusieurs points :

- migrant et hôte doivent s'entendre sur la nouvelle identité du migrant,
- l'hôte pose ses conditions sur les liens que le migrant peut conserver,
- les tiers posent des conditions sur les liens qui peuvent être conservés avec A.

Le premier point est le plus délicat : la solution générale passe par une négociation entre le migrant et l'hôte. Formaliser cette négociation est à la fois indispensable et réducteur.

Le second point est une condition de la migration sur laquelle l'hôte a le dernier mot.

Le troisième point peut être fixé statiquement, ce que nous verrons plus loin.

Nous sommes à présent en mesure de proposer un schéma.

5.3 Proposition

Dans cette proposition nous séparons complètement migration physique et migration de responsabilité.

5.3.1 Gestion des identités

Nous conservons la notion d'identité, ensemble d'identifiants attaché à chaque location. Cependant, nous simplifions le schéma, en attribuant une seule identité par location, c'est-à-dire en supprimant les notions d'identité réelle ou effective, et de location *setuid* ou *non-setuid*.

Nous modifions les règles de propagation de la manière suivante :

- $a[D : P]$: lors de la création par b d'une sous-location a, l'identité I_a est choisie par b, avec pour seule contrainte : $\emptyset \neq I_a \subseteq I_b$. La notion de location *root* est alors implicitement définie : une location *root* est une location possédant dans son identité tous les identifiants de ses administrés,
- lors d'une migration physique, il n'y a aucune modification des identités,
- l'identité peut changer lors d'une migration de responsabilité : si a migre vers b, l'identité de a après migration I'_a vérifie : $\emptyset \neq I'_a \subseteq I_a \cup I_b$; nous en détaillons plus loin le mécanisme.

5.3.2 Mobilité physique

La mobilité physique se fait à l'initiative du migrant, avec acceptation ou refus de l'hôte. Par exemple l'hôte donne une capacité de migration, valable une fois.

5.3.3 Mobilité de responsabilité

Comme nous l'avons vu précédemment, la mobilité de responsabilité comporte deux aspects :

- un aspect conditionnel : une vérification est effectuée sur les liens du migrants,
- un aspect opérationnel : la migration est effective lorsque le changement d'identité a eu lieu.

Aspect conditionnel

L'hôte fournit un ensemble d'identifiants I_{link} qui ont le droit de rester en contact avec le migrant, en lien *IN* ou *OUT*. Pour que la migration de la location a puisse s'effectuer, il faut que :

- pour toute variable libre u de a, définie sur une location x, $I_x \cap I_{\text{link}} \neq \emptyset$,
- pour toute variable u définie sur a et libre dans x, $I_x \cap I_{\text{link}} \neq \emptyset$.

Les tiers interviennent de la manière suivante : une location peut lors de la création d'un canal u définir un ensemble d'identifiants I_u, correspondant aux identifiants qui ont le droit d'hériter de u lors d'une migration. On peut généraliser ceci en considérant que les canaux non contraints ont un ensemble I_u comprenant tous les identifiants possibles. Alors pour que a puisse revêtir l'identité I'_a, il faut que :

- pour toute variable libre u de a, $I_u \cap I'a \neq \emptyset$

Cette manière de gérer les tiers, permet de les faire intervenir de manière passive dans le déroulement de la migration.

Si une seule de ces conditions n'est pas remplie, la migration n'est pas possible.

Aspect opérationnel

La nouvelle identité I'_a du migrant a doit être négociée avec l'hôte b, la seule contrainte étant :
$\emptyset \neq I'_a \subseteq I_a \cup I_b$

Le protocole de négociation n'a pas à être figé, et peut se faire par des communications préalables classiques. Une solution est de réaliser un *join*, c'est-à-dire que la migration se déroule de cette manière :

- le migrant et l'hôte envoient chacun un message système contenant :
  - le nom de la location migrante
  - le nom de la location hôte
  - la nouvelle identité souhaitée du migrant I'_a
  - le filtre I_{link}
  - un canal de retour, portant le résultat
- le système attend de recevoir deux messages identiques (au canal de retour près)
- il effectue alors les vérifications :

  - $\emptyset \neq I'_a \subseteq I_a \cup I_b$,
  - vérifications liées à I_{link},
  - vérifications des conditions des tiers.
- en cas de succès, l'identité de a est remplacée par I'_a.
- un message de résultat est envoyé sur les canaux de retour.

5.4 Remarques

On remarque tout d'abord que la migration de responsabilité peut se faire sans changement d'identité de a : rien n'empêche I'_a d'être égal à I_a. Cette possibilité est intéressante car elle permet à une location de prouver sa bonne foi à une autre, ce qui peut être un préalable à une migration physique.

La protection reste fragile, mais légère : contrairement à la solution de type *unix*, les tests de sécurité n'interviennent que lors des migrations, et non à la réception de chaque message. On y gagne très vraisemblablement en performances. Cependant, le fait de ne pas conserver de lien vers une location douteuse X lors de la migration, n'empêche *a priori* pas d'en tendre un une fois la migration effectuée : il est, pour cela, nécessaire et suffisant d'avoir conservé un lien vers une location susceptible de fournir un lien vers X. Ainsi, l'utilisation de I_{link} se limitera probablement à deux types d'emploi : ou bien I_{link} contiendra tous les identifiants, ou bien il en contiendra très peu.

5.5 Variantes

5.5.1 Liaison des deux mobilités

Une variante consiste à lier mobilité physique et mobilité de responsabilité : on conserve comme seule mobilité possible la mobilité de responsabilité telle qu'elle est décrite plus haut, accompagnée d'une mobilité physique de a sous b.

En adoptant cette solution, on empêche la prise de responsabilité d'une location distante, mais on peut la reconstruire en deux étapes :

1. la location a, sous-location de c, migre sous b, en changeant d'identité

2. une fois là, elle repart sous c, sans changement d'identité, comme cela est décrit dans les remarques précédentes.

5.5.2 Mobilité groupée

Telle que nous l'avons décrite, la mobilité de responsabilité ne concerne qu'une location à la fois. On peut se demander ce qu'il advient des sous-locations du migrant. Cela nous oblige à réfléchir au sens du lien *location – sous-location.*

Dans le join-calcul, l'arbre de location permet principalement de déterminer des ensembles de locations migrant ensemble : une location entraîne tous ses descendants lors de ses déplacements. Ces ensembles n'ont donc rien à voir avec la mobilité de responsabilité. En effet, dans le cas où une location A a fait venir une sous-location X pour des raisons de performances, A souhaitera que X la suive dans ses déplacements. Mais X ne suivra pas A dans ses changements d'identité.

La mobilité physique appelle une hiérarchie physique des locations, où l'on suppose qu'une location et toute sa descendance se trouvent sur le même machine. De même, la mobilité de responsabilité appelle une hiérarchie d'identité, qui donne les ensembles de locations liés par changement d'identité.

On peut alors définir un second arbre de locations, moyennant l'introduction d'une contrainte lors de la mobilité de responsabilité. Cet arbre de responsabilité obéit à la règle suivante :

- si a est le fils de b, alors $I_a \subseteq I_b$

Ainsi, l'arbre est défini de la manière suivante :

- $a[D : P]$: lors de la création par b d'une location a, a est placée en sous-location de b,
- lors d'une migration physique, il n'y a pas de déplacement dans l'arbre des responsabilités
- lors d'une migration de responsabilité de a vers b, a se déplace éventuellement sous b dans l'arbre de responsabilité. En effet il faut modifier la règle $\emptyset \neq I'_a \subseteq I_a \cup I_b$; celle-ci devient :
  - ou bien $\emptyset \neq I'_a \subseteq I_a$, et a ne bouge pas dans l'arbre,
  - ou bien $\emptyset \neq I'_a \subseteq I_b$, et a se déplace sous b.

Chapter 6

Conclusions

Les processus mobiles sont une évolution naturelle de l'informatique. En se déplaçant, le processus gagne en souplesse, car il conserve ses liens. Il gagne en efficacité puisqu'il peut se rapprocher des ressources qu'il utilise. Il gagne aussi en indépendance car il peut travailler successivement pour différents utilisateurs, et changer indéfiniment d'identité.

Ces changements d'identité successifs sont une originalité des processus mobiles, ceci étant habituellement empêché pour des raisons de sécurité : sous unix, seul un processus appartenant à *root* peut changer d'identité, une seule fois. La sécurité des processus mobiles est donc un problème plus compliqué que dans les systèmes classiques et notamment dans les systèmes de code mobile.

Les méthodes de sécurité fondées sur la cryptographie sont utiles pour résoudre les questions d'implémentation et celles d'identification des utilisateurs. En faisant quelques hypothèses raisonnables sur l'environnement des processus mobiles, on peut se ramener dans un cadre théorique clair, où l'identification des utilisateurs et l'authentification des messages est réalisée par le système d'exploitation.

Dans ce cadre clair, la question de la sécurité prend une autre dimension. D'une part, il est possible de protéger de manière dynamique l'accès aux ressources, en installant par exemple un système de type unix, où les fichiers sont des canaux. Cependant ce système seul est relativement lourd, et réagit mal aux changements répétés d'identité des processus.

D'autre part, il est possible de contrôler les liens d'un processus, c'est-à-

dire d'analyser les identités des processus avec lesquels il communique. Une telle vérification est une garantie forte qu'un processus peut présenter à son futur propriétaire comme gage de bonne foi lorsqu'il s'apprête à changer d'identité.

La mobilité physique est indissociable d'une mobilité concernant l'identité d'un processus et que nous appelons mobilité de responsabilité. En examinant la question, il apparaît que les deux mobilités sont à la fois indissociables et indépendantes. Lier l'une à l'autre est arbitrairement réducteur. Dès lors, les structures de la mobilité physique sont duplicables et applicables à l'autre mobilité : on définit ainsi un second arbre des locations relatif à la responsabilité et non à la géographie.

Les solutions présentées ici s'efforcent de rester générales. Force est de constater qu'elles ne sont ni simples, ni élégantes, notamment en ce qui concerne la gestion des identités. Il serait intéressant de chercher une solution a priori réductrice, mais permettant en fait de réaliser les mêmes fonctions, de manière simple.

Remerciements

Je tiens à remercier Jean-Jacques Lévy qui m'a accueilli dans son équipe et guidé patiemment tout au long de ce stage.

Je tiens également à remercier toute l'équipe du projet PARA, et notamment : Damien Doligez, Cédric Fournet, Georges Gonthier et Thérèse Hardin. Leur accueil fut des plus chaleureux, et nos nombreuses discussions fort enrichissantes.

Mes remerciements vont finalement à Robert Cori et à Jacques Stern qui m'ont autorisé à faire un stage qui dépasse le cadre strict du DEA d'algorithmique.

Bibliography

[1] M.Abadi, M.R. Tuttle. [1991] *A Semantics for a Logic of Authentication*

[2] M.Abadi, R.M. Needham. [1995] *Prudent Engineering Practice for Cryptographic Protocols*

[3] M.Abadi, A.Gordon. [1996] *Pi-Calculus and Cryptography*

[4] G.Berry, G.Boudol. [1992] *The Chemical Abstract Machine*

[5] J.Berstel, J.F.Perrot. [1986] *MULTICS : guide de l'usager*

[6] M.Burrows, M.Abadi, R.M. Needham. [1989] *A Logic of Authentication*

[7] M.Burrows, M.Abadi, R.M. Needham. [1990] *The Scope of a Logic of Authentication*

[8] H.Cluster. [1993] *Inside Windows NT*

[9] G.Coulouris, J.Dollimore, T.Kindberg. [1994] *Distributed Systems : concepts and design*

[10] D.Dean, D.Wallach. [1995] *Security Flaws in the HotJava Web Browser*

[11] C.Fournet, G.Gonthier. [1996] *The reflexive CHAM and the join-calculus*

[12] C.Fournet, G.Gonthier, JJ.Lévy, L.Maranget, D.Rémy. [1996] *A calculus of mobile agents*

[13] D.Hagimont, J.Mossière, X.Rousset de Pina, F.Saunier. [1996] *Hidden Software Capabilities*

[14] B.Lang, C.Queinnec, J.Piquer. [1992] *Garbage Collecting the World*

[15] Leffler, McKusick, Karels, Quaterman. [1989] *4.3BSD Unix Operating System*

[16] H.Levy. [1984] *Capability-based computer systems*

[17] R.Milner. [1991] *The Polyadic π-Calculus : a Tutorial*

[18] S.Mullender, A.Tanenbaum. [1984] *The Design of a Capability-Based Distributed Operating System*

[19] Sophie L.Poirot-Delpech. [1995] *Biographie du Cautra*

[20] RL.Rivest, A.Shamir, L.Adleman. [1978] *A Method for Obtaining Digital Signatures and Public-key Cryptosystems*