Analyse des langages et modèles de la mobilité

Gérard Boudol, Florence Germain, Marc Lacoste

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Analyse des Langages et Modèles de la Mobilité

Gérard Boudol — Florence Germain — Marc Lacoste

N° 3930

Avril 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3930--FR

Analyse des Langages et Modèles de la Mobilité

Gérard Boudol , Florence Germain , Marc Lacoste

Thème 1 — Réseaux et systèmes
Projet Mimosa

Rapport de recherche n° 3930 — Avril 2000 — 73 pages

Résumé : Ce document présente une analyse comparative de divers langages prototypes et modèles formels existants pour la mobilité de code. Trois aspects sont retenus pour cette étude: la distribution, la mobilité, et la sécurité. L'étude est également structurée autour de la notion de domaine qui nous semble centrale dans ces langages et modèles, et illustre différentes sémantiques de cette notion.

Mots-clés : code mobile, mobilité de processus, calculs de processus

Ce travail a été partiellement soutenu par le projet RNRT Marvel. F. Germain et M. Lacoste travaillent à France Télécom-Recherche et Développement, Meylan, France.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 —Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Models and Languages for Mobility

Abstract: This document contains an analysis of various prototypical languages and formal models for mobile code. We focus on the distribution, mobility and security aspects. The analysis also shows that the notion of domain seems to be a primitive concept underlying these languages and models. Besides, it illustrates various semantics for this notion of domain.

Key-words: mobile code, mobile processes, process calculi

Introduction

Ce document présente une analyse comparative de divers langages et modèles de la mobilité, effectuée dans le cadre du projet RNRT MARVEL. Cette analyse de l'existant n'est pas exhaustive: nous avons en effet été amenés à faire des choix en ce qui concerne le matériel étudié, pour diverses raisons. Une première raison est l'insuffisance, dans certains cas, de la documentation disponible. Cela concerne en particulier les "produits du commerce", comme Odyssey, Aglets, ObjectSpace Voyager..., qui sont tous constitués de bibliothèques de classes JAVA permettant dans une certaine mesure de programmer la migration d'agents, pour lesquels les documents immédiatement accessibles sont essentiellement de la publicité. Il faut noter ici que les limitations du langage JAVA – on ne peut "sérialiser", c'est à dire mettre sous une forme transmissible, que certains types d'objets, et en particulier on ne peut capturer de cette façon l'état d'un "thread" en cours d'exécution –, et plus généralement des langages où tout est objet, ne permettent pas de mettre en œuvre l'idée d'agent migrant dans toute sa généralité, telle qu'elle avait été proposée initialement par le manifeste Telescript par exemple. Dans la mesure où l'un des objectifs du projet MARVEL est d'aller au delà de ces limites, nous pensons que l'absence de notre analyse de ces extensions de JAVA ne pénalise pas notre travail.

D'un autre côté, nous avons dû faire aussi un choix parmi les langages et modèles dont la description est suffisamment précise pour se prêter à l'analyse, tout simplement parce que faire une revue détaillée visant à l'exhaustivité nous aurait entrainé trop loin. Nous pensons néanmoins que l'inventaire choisi – qu'on ne doit d'ailleurs pas considérer comme clos – est assez représentatif: on verra qu'il présente déjà une grande variété de propositions, illustrant le fait que les choix techniques sont loin d'être résolus concernant le code mobile. Nous nous sommes essentiellement limité à des propositions qui traitent de *migration*. Pour nous ceci veut dire que le modèle ou le langage comporte une notion – parfois implicite, souvent minimale – de "site", c'est à dire de répartition spatiale, par rapport à laquelle s'effectue la migration. Ceci exclut a priori les modèles où la *mobilité* consiste en la transmission de "programmes", ou d'accès à ces programmes, soit à travers des canaux de communication, soit directement comme arguments d'autres programmes. Le représentant le plus fameux de ce type de mobilité est le π-calcul, qui est à la base de bon nombre de modèles et langages que nous étudions, mais que nous ne considérons pas comme un modèle de la migration proprement dite. Dans la suite du document lorsque nous parlons de mobilité il faut donc comprendre plus précisément: migration, ceci impliquant une idée de site.

Notre étude envisage les langages et modèles sous trois aspects: la *distribution*, *mobilité* et *sécurité*. Il y a évidemment d'autres dimensions à envisager, notamment du point de vue de l'implémentation, et ceci fait l'objet de travaux complémentaires dans MARVEL. Un "élément fédérateur" pour l'ensemble des caractéristiques que nous étudions est l'existence de "groupes" d'entités (entités de nature passive comme les ressources, ou de nature active comme les processus ou les agents) partageant une "unité comportementale", et respectant vis à vis des autres groupes une certaine "indépendance mutuelle". La notion de site fournit un exemple typique de tel "groupe", qui peut être une unité d'exécution (un interprète par exemple), une unité de communication (un espace d'adressage), de défaillance, de sécurité, d'administration ou encore de migration. Notre analyse sera donc structurée pour une large part par ce que nous appellerons *domaine*, incluant en particulier la notion de site. Cette notion nous semble importante, bien qu'encore imparfaitement mise à jour, en particulier comme concept de programmation.

Le document est structuré de la manière suivante: une première partie explicite la grille d'analyse retenue. Cette partie est découpée en trois rubriques, présentant respectivement les trois axes d'analyse retenus, à savoir la distribution, la mobilité et la sécurité. La deuxième partie se présente sous la forme d'une suite de fiches dédiées aux principaux modèles et langages étudiés – on en trouve la liste dans la table des matières. Les fiches sont identiques dans leur structuration et suivent fidèlement le schéma d'analyse présenté dans la première partie du document. Bien sûr, certains items de la grille d'analyse pourront, le cas échéant, ne pas avoir de contenu (ou tout simplement pas de sens) pour un langage ou un calcul donné; dans ce cas *ces items seront*

omis de la fiche correspondante. Le dernier chapitre du document présente quelques conclusions de notre étude, et quelques perspectives.

Première partie

Critères d'Analyse

Chapitre 1

Distribution

Notre analyse au sens de la distribution s'articule autour du concept de domaine. Plus précisément, nous considérons que la distribution *est* l'existence de domaines. Un domaine regroupe des processus (ou des agents) individualisés par une unité de comportement. Dans l'état actuel de l'art, un calcul ou un langage manipule en général une seule sémantique de domaine, et ce de manière totalement explicitée (cas de la plupart des calculs) ou semi-implicite. La sémantique de domaine varie selon le langage/calcul considéré: sémantique orientée vers la co-localisation physique, la défaillance, l'administration, la sécurité...

Les critères de comparaison retenus dans le présent chapitre portent principalement sur la sémantique des domaines, la structure de l'espace des domaines, la communication et la désignation intra- et inter-domaines. On aborde de manière plus succincte la problèmatique des défaillances.

1.1 Concept de domaine

L'ensemble des calculs de processus orientés vers la distribution intègrent un certain nombre de manifestations de domaines, sans pour autant réifier véritablement le concept de domaine. Seul le calcul des ambients propose une véritable primitive de domaine (appelée ambient). La première de ces manifestations est le nom de domaine, introduit par l'ensemble des calculs. D'autres manifestations sont exploitées, différant selon les calculs: le "processus de localité" ($\pi_{1\ell}$), le processus localisé ($\pi_{1\ell}$), la solution localisée (Join), la définition localisée (Join), la configuration ($\pi_{1\ell}$), la solution distribuée (Join), le "process body" (Seal) ...

Le fait qu'un domaine soit nommé permet d'assimiler un domaine à une ressource. Un domaine peut donc être désigné, en tant que cible d'une communication distante par exemple, ou encore en tant que cible d'une migration de processus ou d'une migration de domaine. L'ensemble des calculs s'accordent à considérer qu'un domaine "abrite" des processus. Parfois ($\pi_{1\ell}$), le domaine est lui-même assimilé à un processus particulier jouant le rôle de processus de contrôle du domaine.

La sémantique de domaine prend, selon les modèles, diverses orientations selon la vocation première du modèle: sécurité (Ambients, Seal), défaillances ($\pi_{1\ell}$, Join), communication (Dπ, Ambients), migration (Join, Ambients, Seal) ... Chacun des calculs étudiés prévoit une sémantique de domaine unique, laquelle capture en général plusieurs orientations. Ainsi, par exemple, les domaines du Join sont à la fois des domaines de défaillance et de migration.

1.1.1 Topologie de l'espace des domaines

L'espace des domaines introduits par les calculs étudiés est le plus souvent structuré hiérarchiquement (Join, Ambients, Seal), sous forme d'arbre ou sous forme de forêt. Dans le $\pi_{1\ell}$-calcul, on retrouve indirectement une forme de hiérarchie au sens où l'espace de domaines (appelés dans le $\pi_{1\ell}$ "localités") constitue l'ensemble des feuilles d'une arborescence de configurations. Cependant, dans ce calcul, l'ensemble des domaines du $\pi_{1\ell}$-calcul en lui-même est plat.

Le fait de structurer les domaines sous la forme d'une arborescence permet d'expliciter assez naturellement la "migration" des domaines les uns par rapport aux autres, sous la forme d'une reconfiguration locale de l'arborescence. Nous désignerons cette forme de migration par le terme de "migration topologique", pour désigner le déplacement d'une sous-arborescence dans l'arborescence (Join, Ambient, Seal). Dans une arborescence de domaines, chaque noeud désigne un domaine, abritant ses propres processus et possédant un certain nombre de domaines fils.

Certains des calculs sous-entendent l'existence d'un éther entre les domaines (pil-calcul). L'éther peut accueillir des messages en cours d'acheminement d'une localité à une autre. Le concept d'éther est un élement important de la formalisation de l'asynchronisme de la communication distante.

1.1.2 Sémantiques de domaine

Ce paragraphe explicite les diverses sémantiques de domaines rencontrées, et leur conséquence en terme de pré-requis d'infrastructure.

- Orientation "communication": les entités d'un même domaine sont unifiées par un mode de communication locale, i.e. intra-domaine, même si le domaine peut être réparti sur plusieurs espaces d'adressage au sein d'une même machine, ou encore sur des machines distantes. Ainsi, certains des calculs étudiés (Ambients, Dπ, Nomadic Pict) font apparaitre les domaines comme des unités de communication, au sens où la communication directe d'un domaine à un autre n'est pas possible.
- Orientation "mobilité": les entités d'un même domaine sont unifiées par leur comportement en terme de migration, au sens où le calcul prévoit une primitive de migration atomique (indivisible) du contenu d'une localité. Lorsque l'espace des localités est structuré de manière arborescente, la migration d'une localité s'accompagne de la migration de toutes ses localités filles. C'est le cas pour le Join, les Ambients et le Seal. Cette interprétation est traitée dans le chapitre "mobilité".
- Orientation "defaillance": les entités d'un même domaine sont unifiées par un même comportement vis-à-vis de la défaillance (toutes sont défaillantes, ou aucune). La défaillance d'une localité est traduite en terme d'incapacité à émettre et/ou recevoir des messages ou des localités migrantes. Cette hypothèse facilite l'explicitation dans le calcul de protocoles de détection, et de contrôle, des défaillances. Cette orientation est délibérément prise dans le $\pi_{1\ell}$ et le Join. Cette interprétation fait l'objet du paragraphe "Domaines et défaillances".
- Orientation "sécurité": les entités d'un même domaine sont unifiées par un même protocole d'accès (toutes sont accessibles, ou aucune). L'ensemble des calculs étudiés ($\pi_{1\ell}$, Join, Ambients, Seal) intégrent un concept minimal de sécurité via la gestion de la portée des noms (noms de canaux, noms de domaines). Au-dela de cette vue, certains calculs rendent explicites les concepts de frontière et d'autorisation d'entrée dans un domaine (Ambients), ou encore d'autorisation d'accès à une ressource hébergée par un domaine (Seal). Cette interprétation est abordée dans le chapitre "sécurité".

1.1.3 Observabilité/contrôlabilité des domaines

Il s'agit de l'aptitude du calcul à permettre l'observation et/ou le contrôle des domaines depuis l'application. L'observation/contrôle de domaine est en général liée à la notion de défaillance ($\pi_{1\ell}$, Join). Il peut alors s'agir de contrôler un état, lorsque la notion d'état est explicitée (tel est le cas dans le $\pi_{1\ell}$ par exemple, où le domaine est très étroitement associé à un processus de contrôle à état). Mais la défaillance n'est pas la seule motivation pour le contrôle de domaines. Ainsi, dans les Ambients, le contrôle est motivé par des considérations de sécurité, et il prend la forme d'une dissolution de la "frontière" de l'ambient (ce qui a pour conséquence la levée des droits d'accès aux processus considérés). Une forme encore différente de contrôle se manifeste par les domaines gardés: domaines gardés par une capacité à migrer dans le calcul des Ambients, domaines gardés par des patterns de noms joints en Join. Dans ce cas, c'est la consommation de la capacité, ou encore l'activation du pattern Join, qui rend "actif" le domaine gardé. Enfin, une dernière forme de contrôle apparaît dans le Join, où il est possible d'activer/inhiber dynamiquement une localité en exploitant le concept de définition (forme endémique de localité).

1.2 Domaines et communication locale

Le terme de communication "locale" désigne ici la communication intra-domaine (par opposition à la communication "distante", inter-domaines).

1.2.1 Désignation locale

Il s'agit du problème de désignation d'une ressource (typiquement, un récepteur) par un processus appartenant au même domaine que la ressource. Le nom de la ressource doit être syntaxiquement "connu" du processus

qui la désigne (au sens où ce processus doit être présent dans la portée du nom en question). Un tel nom peut avoir été acquis par la communication. Dans un même domaine, un nom peut désigner un processus récepteur unique ($\pi_{1\ell}$), ou un ensemble de processus récepteurs localisés sur le domaine en question (Join, Seal). La désignation est directe, c'est à dire basée sur le nom primitif. Dans le Seal, la désignation locale s'accompagne d'une dénotation du nom de la ressource, explicitant le caractère local de la chose.

1.2.2 Politique de communication locale

Il s'agit de la communication à l'intérieur d'un même domaine. Dans tous les calculs étudiés (sauf le Seal), la communication locale est asynchrone, polyadique et "unicast" (deux processus ne peuvent pas recevoir sur un même nom de façon atomique). La communication locale dans le Seal est synchrone et "unicast". Elle peut être nommée (pil, Join, Seal) ou anonyme (ambients).

Tous les calculs étudiés peuvent au minimum communiquer des noms de canaux et des noms de domaines. Le calcul des Ambients prévoit la communication d'entités plus "précises" que les noms, à savoir les capacités (une capacité pouvant être comprise comme un usage spécifique sur un nom). Dans tous les calculs étudiés, la réception est statique, c'est à dire que la communication de noms ne conduit pas à l'appropriation de ces noms (au sens de récepteurs "propriétaires") par le processus qui reçoit le nom. Un processus qui reçoit un nom de canal acquiert le droit d'émettre à destination de ce nom. Dans le même ordre d'idées, un processus qui reçoit un nom de domaine acquiert le droit à désigner le domaine en question à des fins de migration (Join, ambient), d'observation ou de contrôle ($\pi_{1\ell}$).

1.2.3 Sémantique de la réception locale

La réception peut être unique (un seul processus récepteur a la capacité de recevoir sur un nom) ($\pi_{1\ell}$) ou multiple (Join). Dans le cas du Join, plusieurs processus récepteurs sur un même nom sont nécessairement co-localisés. Ce point est important du point de vue de l'implémentation, puisqu'il garantit le caractère local du mécanisme de décision d'infrastructure sous-jacent. Notons au passage le fait que plusieurs ambients peuvent "recevoir" sur un même nom, ce qui va dans le sens d'une identification entre un ambient et un processus du $\pi_{1\ell}$.

Dans la plupart des calculs étudiés (à l'exception de Nomadic Pict et de Seal), la réception est statique, au sens où l'appropriation de récepteurs par la communication est impossible: on ne peut créer de récepteur pour un nom que l'on reçoit. Enfin, dans tous les calculs, un nom reçu devient "connu", au sens où il devient référençable.

1.3 Domaines et communication distante

1.3.1 Désignation distante

Il s'agit de la désignation d'une ressource (le cas échéant un domaine) par un processus appartenant à un domaine distant. De même que pour la désignation locale, le nom de la ressource doit être syntaxiquement "connu" du processus qui entend la désigner. Tous les calculs étudiés intégrent la notion de portée distribuée (la portée d'un nom peut s'étendre au système entier). Ceci permet de référencer une entité du calcul par son nom depuis une localité distante.

La désignation distante peut être globale (tout est désignable depuis un quelconque domaine) ($\pi_{1\ell}$, Join) ou restreinte (seule la désignation de "voisinage", au sens de l'arborescence, est possible) (Ambients, Seal). Enfin, la localisation peut être absolue ($\pi_{1\ell}$, Join, Ambients) ou relative par rapport à l'entité désignante (Seal).

1.3.2 Politique de communication distante

De même que pour le désignation, la communication distante peut être globale (on peut communiquer de façon atomique, de n'importe quel domaine vers n'importe quel domaine) ($\pi_{1\ell}$, Join) ou restreinte (seule la communication de "voisinage" est possible) (Ambients, Seal, Nomadic Pict), voire même purement locale (à l'intérieur du même domaine – Dπ). Dans tous les calculs étudiés sauf le Seal, la communication distante est asynchrone: un message est émis, et "plus tard" il est reçu.

La localisation cible peut être implicite ou explicite dans le message[1]. Le transport du message peut être implicite (un message est émis à destination d'un nom, puis il est reçu sur ce nom), ou explicite (un message émis à destination d'un nom doit d'abord sortir du domaine source, puis pénétrer dans le domaine cible, puis interagir avec le nom). Ainsi, le $\pi_{1\ell}$ prévoit une communication distante en 2 phases (sortie du domaine courant, puis intrusion-consommation atomique). Le Join capture de même deux phases différentes (sortie-entrée atomique, puis consommation).

1.3.3 Routage logique

On appelle "routage logique" l'acheminement dans l'arborescence des domaines. Ce routage capture effectivement la notion de routage dans un réseau lorsque les domaines ont une sémantique de communication.

Le routage des messages est explicite dans le calcul des ambients, Dπ, Nomadic Pict et dans le Seal, dans lequel l'interaction primitive est restreinte au voisinage dans l'arborescence. Dans les autres calculs étudiés, la localisation cible est totalement transparente dans le cadre de la communication distante[2].

1.4 Domaine et défaillances

Certains des calculs étudiés ($\pi_{1\ell}$ et Join) ont leur sémantique fondée sur le concept de défaillance partielle. Il s'agit pour ces calculs de représenter de façon primitive un groupe de processus liés par un même comportement défaillant, au sens du blocage temporaire ou définitif de certaines de leur capacités vis à vis de l'émission/réception de messages ou vis-à-vis de la mobilité. Dans les deux cas, une localité défaille de manière atomique, en compagnie de l'intégralité de ses localités filles. Une telle défaillance est détectable depuis l'extérieur de la localité et contrôlable depuis l'extérieur ($\pi_{1\ell}$) ou depuis l'intérieur (Join) de la localité. Du point de vue de la représentation, le statut de localité en panne peut être capturé par un état particulier ($\pi_{1\ell}$), ou par un simple label (Join). En Join, une localité arrêtée prend le statut d'une définition (forme endémique de localité). Dans les deux cas, une localité arrêtée est supposée capable de répondre à un message de test de défaillance.

1. Dans le cas de messages de contrôle à destination d'un domaine, la localisation cible est nécessairement explicite, puisque le nom du domaine est aussi le nom du canal sur lequel le message est envoyé. Cela sous-entend quand même que les noms de domaines sont uniques à l'échelle globale (d'où la nécessité d'un mécanisme d'infrastructure afférent).

2. Le cas d'un routage transparent nécessite la mise à disposition d'un mécanisme distribué d'infrastructure permettant de gérer cette transparence.

Chapitre 2

Mobilité

L'analyse de la mobilité dans les langages et les calculs s'organise autour de trois notions:

- les **domaines**. On ne s'interesse ici qu'à l'aspect "mobilité" des domaines, dont d'autres aspects sont analysés dans la première partie du document. Un domaine est donc ici ce qui sert de référence à la mobilité: le domaine est ce que l'on quitte et ce vers quoi l'on va lorsqu'on fait acte de migration. Du point de vue de la mobilité, cette notion est souvent implicite, ou réduite à celle d'adresse (Internet par exemple).
- les **entités migrantes**. Nous n'étudions pas des langages ou modèles, de type client/serveur (RPC ou RMI), qui seraient fondés sur la seule migration de données. Les entités migrantes auront donc en général un contenu exécutable, mais celui-ci peut prendre des formes assez diverses (domaines, objets, clôtures...).
- le **contexte**. C'est tout ce qui constitue l'environnement de l'entité mobile au moment de sa migration. Cette notion reste implicite dans les calculs de processus, tandis que la gestion du contexte au moment de la migration est une part importante de l'implémentation des langages pour la mobilité.

Ces notions ne sont pas disjointes: dans certains modèles les domaines sont les entités migrantes, et dans d'autres l'entité migrante inclut – ou plus exactement entraine – une portion de contexte. En général, une entité migrante est contenue dans un domaine, tandis que son contexte peut s'étendre sur plusieurs domaines. Dans le modèle traditionnel de la sémantique, le "contexte" est constitué de l'*environnement*, qui associe à chaque identificateur une "valeur désignable" (celle-ci pouvant être en particulier une adresse mémoire), et de la "*mémoire*" ("store"), qui associe une valeur à chaque adresse. En particulier chaque élément du contexte est nommé, par un identificateur ou une adresse, et l'on peut dire que le contexte d'une entité est la portion d'environnement et de mémoire dont l'entité connait (par l'occurence d'un identificateur libre), de manière transitive, le nom. On pourrait en fait avoir une vision plus large de ce qu'est le contexte d'une entité, incluant par exemple les dispositifs physiques ou les bibliothèques de programmes sur lesquels une unité d'exécution peut avoir un lien, mais ceci n'est généralement pas le cas dans les modèles et langages étudiés.

On voit souvent faite la distinction entre migration **forte** et mobilité **faible**: dans la première forme, une "unité d'exécution" peut être interrompue, envoyée avec son état courant vers un autre site, et relancée à son arrivée, tandis que dans la mobilité faible ce n'est pas une unité d'exécution qui migre, mais seulement du code exécutable. Cette distinction n'est pas assez fine pour analyser les modèles et langages que nous étudions; nous la raffinerons donc par la suite.

On peut distinguer aussi deux types de mobilité: la mobilité par **déplacement**, dont la sémantique est évidente, et la mobilité par **réplication**, où un "clone" de l'entité migrante se déplace, tandis que son "double" reste dans le domaine de départ. En fait cette distinction s'applique principalement aux éléments du contexte de l'entité migrante. On trouve néanmoins quelques exemples de migration de code procédant par réplication: le plus connu est le "code on demand" à la JAVA, qui doit être vu comme un "cas limite" de mobilité[1], que l'on trouve aussi dans Obliq; une autre forme se trouve dans Agent Tcl, par clonage d'un agent à distance.

Cardelli et Gordon ont mis en évidence une distinction que l'on peut faire dans l'expression de la mobilité: le mouvement peut être "**objectif**" ou "**subjectif**"; dans le premier cas l'instruction de migration se trouve hors du contrôle de l'entité migrante, et dans le second au contraire cette instruction provient "de l'intérieur". Cette distinction opère un clivage entre les modèles de la migration de domaine, où elle peut s'appliquer utilement, et

1. on rappelle qu'un des objectifs annoncés du projet MARVEL est d'aller au delà de JAVA en matière de mobilité. Nous n'étudions donc ici que les propositions de langages ou de modèles qui vont dans ce sens.

ceux de la migration de code, où il semble que la mobilité s'exprime toujours de manière objective. Toutefois nous l'utiliserons, principalement pour qualifier un "style". Dans le même ordre d'idée, nous analyserons le caractère que peut avoir la migration par rapport à l'exécution du migrant. En effet la migration peut être **bloquante** vis à vis de l'entité migrante – c'est le cas typiquement dans la migration de "thread" ou de continuation – ou **asynchrone**, comme dans le cas de migration d'agents ou de domaines, qui peuvent recevoir des communications en concurrence avec l'instruction de migration.

Une autre distinction doit être faite à propos du contexte: celui-ci comporte nécessairement une partie **publique** – au minimum un serveur de noms, mais plus généralement un ensemble de bibliothèques nécessaires à l'exécution du code migrant – et éventuellement une partie **privée**, dont la portée est délimitée. Il semble que dans tous les cas, le comportement de l'entité migrante par rapport à la partie publique de son contexte soit celui du "dynamic linking": la valeur d'un nom public est celle qu'il a dans le site d'exécution courant, et il n'y a pas de migration de valeur publique. D'un autre côté, tous les modèles étudiés reposent, en ce qui concerne les noms privés, sur la discipline de la portée lexicale ("static binding"). Nous ne reviendrons donc pas sur la distinction public/privé (ni "static/dynamic binding") qui n'a pas de vertu "classificatoire" pour notre étude.

Il faut noter que "privé" ne veut pas dire "exclusif": certains éléments du contexte, dont la portée est connue, peuvent être partagés entre plusieurs entités migrantes – ceci si le modèle autorise le parallélisme, même fictif (par exemple par un système de "threads", avec gestion de l'ordonnancement), entre ces entités. Dans ce cas la mise en œuvre de la mobilité doit naturellement gérer cette situation. On peut distinguer plusieurs manières de gérer les liens qu'entretient une entité migrante avec son contexte, et particulièrement au moment où la migration intervient, en distinguant plusieurs types de comportement du contexte par rapport à l'entité migrante:

- **contexte fixe**. Ici les éléments du contexte restent dans le domaine d'où émane le migrant. Deux sous-cas se présentent en ce qui concerne la relation du migrant avec son contexte:
  - la relation est rompue. Ici il peut se faire qu'un lien soit rétabli dans le domaine destinataire, par "dynamic linking", avec une ressource de même nom ou de même type que celle que l'on quitte.
  - la relation est maintenue à distance. Ceci peut se faire de manière soit implicite (une requête du migrant vers son contexte rejoignant spontanément ce dernier), soit explicite, la relation étant assurée par l'intermédiaire d'un "délégué" ("proxy") installé dans le domaine d'arrivée[2].
- **contexte mobile**. On peut distinguer deux sous-cas, selon le type de mobilité présenté par les éléments du contexte:
  - mobilité par déplacement. Si l'élément de contexte est partagé entre l'entité migrante et d'autres unités d'exécution, on retrouve pour ces dernières une situation symétrique de la précédente (lien rompu ou maintenu d'une certaine façon).
  - mobilité par réplication. L'entité migrante emporte une copie d'une partie du contexte. Ici on pourrait encore distinguer le cas où le contexte original et sa copie doivent, quelle que soit leur évolution, rester identiques (cohérence) ou non. Toutefois l'idée de cohérence parfaite est, dans un univers asynchrone, peu plausible; pour l'assurer on aurait plutôt recours à un "proxy" – c'est à dire qu'on serait plutôt dans l'un des sous-cas précédents.

Naturellement cette classification s'applique à chaque élément du contexte plutôt qu'au contexte globalement. Elle est utilisée dans les points 3 et 4 de la "grille d'analyse" que nous présentons maintenant. Les critères retenus pour analyser les modèles et langages en ce qui concerne la mobilité sont les suivants:

1. **nature des entités migrantes**. Il peut s'agir de domaines, d'objets, de code sous diverses formes, "threads" ou clôtures par exemple[3]. Il s'agit surtout ici d'indiquer le "style" du modèle étudié, car la nature exacte (pour autant qu'elle soit connue!) de ce qui migre est précisée dans les points suivants.
2. **expression de la migration**. Nous analysons ici la forme (qui peut être complètement implicite, comme dans le cas du "code on demand") sous laquelle s'opère la migration, et ses caractéristiques par rapport à l'exécution du migrant. On trouve ici les distinctions migration objective/subjective, asynchrone/bloquante (par rapport à l'entité migrante).
3. **mobilité en cours d'exécution**. On distingue ici les modèles où la migration d'une entité exécutable a lieu en préalable à son exécution de ceux où une instruction de migration peut affecter l'entité migrante elle-même au cours de son exécution. Dans la première catégorie on trouve les paradigmes "code on

2. ce cas peut à vrai dire se confondre avec le précédent, si la ressource avec laquelle on établit une nouvelle liaison est justement un "délégué". Une différence est que ce dernier est supposé être installé lors de la migration, pour mettre en œuvre une référence distante.

3. comme on l'a dit, on ne s'intéresse ici qu'aux cas où l'entité migrante a un contenu exécutable.

demand" (COD) et "remote evaluation" (REV), tandis que dans la seconde on trouve le modèle agent, la migration de "thread" ou de domaine.

4. **migration d'un état associé**. Elle a lieu lorsque la migration d'une entité, un "thread" par exemple, provoque celle de tout ou partie de l'état de la mémoire auquel l'entité a accès au moment de la migration. Si on a une telle migration de l'état, associée à la mobilité en cours d'exécution, on parle alors d'agents migrants.
5. **entretien de références distantes**. Lorsqu'une partie du contexte d'exécution de l'entité migrante ne peut pas ou ne doit pas être déplacée ou copiée, on met en place un mécanisme qui permet au migrant de maintenir un lien avec cette partie à travers le réseau (i.e. d'un domaine à l'autre). On parle alors de "calcul global" (Cardelli).

Il y a de nombreuses combinaisons possibles entre les différentes options que l'on peut prendre concernant les points ci-dessus. Toutes n'ont pas été expérimentées. Une voie qui a été largement explorée est la mobilité dans les langages à objets, et particulièrement sous la forme de développements à partir de JAVA.

Dans un langage à objets, la notion d'état et de contexte est assez claire, dans la mesure où chaque objet encapsule son état (il n'y a donc pas de partage possible – si l'on ne fait pas migrer des "threads"), et où l'on peut dire que le contexte est constitué des objets qu'il connait, i.e. dont il peut invoquer les méthodes. La plupart des extensions de JAVA vers la mobilité implémentent donc une forme de mobilité où les entités migrantes sont des objets. La migration peut être "objective" ou "subjective" (infligée à soi-même). L'objet migrant emporte avec lui son état et conserve des références distantes vers les autres objets, éventuellement par l'intermédiaire de "proxies" (la communication se faisant par invocation de méthode). Afin d'offrir un certain caractère de mobilité forte, i.e. de mobilité au cours des calculs, l'objet migrant exécute à son arrivée à destination une méthode particulière, qui peut être un paramètre de l'instruction de migration. Toutefois, dans la mesure où l'implémentation de JAVA ne permet pas la "sérialisation" – donc la mobilité – des "threads", le caractère de "mobilité forte" est assez limité.

Ce scenario est par exemple mis en œuvre, avec des variantes, dans les propositions Telescript/Odyssey (Telescript a "lancé" l'idée d'agent mobile, allant de place en place en collectant des informations, et en effectuant des opérations pour le compte d'un utilisateur; cette idée n'est que partiellement mise en œuvre dans Odyssey), Aglets, ObjectSpace Voyager. On ne trouvera pas dans ce document de fiche spécifique sur ces extensions de JAVA, parce que la documentation dont nous disposons n'est pas suffisamment explicite pour permettre une analyse détaillée (il faudrait expérimenter les différents logiciels pour comprendre la sémantique).

Chapitre 3

Sécurité

On utilisera dans ce chapitre les termes suivants:

- La **sécurité** sera vue comme la possibilité pour un système de protéger des objets en termes de confidentialité et d'intégrité. Le volet disponibilité (possibilité d'accéder à ou d'utiliser des objets ou ressources) qui est aussi traditionnellement associé avec la sécurité des systèmes ne sera pas traité ici: cet aspect semble relever plus de la sûreté de fonctionnement que de la sécurité proprement dite. Le domaine de la sûreté dépasse le cadre de ce chapitre.
- La **confidentialité** sera vue comme la protection d'un objet contre la divulgation non-autorisée d'information et l'utilisation non-autorisée de ressources.
- L'**intégrité** sera vue comme la protection contre la modification non-autorisée de ressources.

Deux acteurs sont essentiels dans la sécurité d'un système:

- Les **objets**. Ce sont des composants passifs comme des ressources ou de l'information.
- Les **entités**. Ce sont des composants actifs comme un utilisateur ou un processus qui vont agir sur les objets. Ils sont dotés de caractéristiques comme une identité ou des privilèges. Cette notion est capturée par le terme anglais "**principal**".

Une distinction importante doit être faite entre **politique de sécurité** et **mécanismes** ou **fonctions de sécurité**.

- Une **politique de sécurité** correspond à des besoins utilisateur en matière de sécurité. Elle peut être définie par un administrateur indépendamment des mécanismes qui l'implémentent. Cette politique de sécurité peut être représentée de manière formelle par un **modèle de sécurité**: certains des calculs étudiés dans ce document permettent l'expression d'un modèle de sécurité e.g. le spi-calcul ou le seal calcul.
- Les **mécanismes de sécurité** permettent l'application d'une politique de sécurité particulière, dans les limites fixées par le pouvoir expressif d'un modèle de sécurité. Ces mécanismes peuvent être offerts conjointement au niveau d'un langage et du run-time qui l'implémente lorsque ce langage fournit explicitement des primitives de sécurité (Pict, Obliq, Java). Ils peuvent aussi être fournis seulement au niveau d'un run-time.

Plusieurs problèmes de sécurité peuvent se poser dans un système à objets mobiles. Il convient d'examiner dans cette architecture les endroits où peuvent se concentrer les attaques pour en déduire les mécanismes de protection appropriés. Pour simplifier, on modélise ici un système à objets mobiles comme comprenant trois éléments:

- Une **structure d'accueil** comprenant une machine, un système d'exploitation et un run-time qui abrite un certain nombre de ressources. Cette structure d'accueil est apparentée à la notion de contexte définie dans le chapitre sur la mobilité ("environnement de l'entité mobile au cours de sa migration").
- Les **unités d'exécution** capables de migrer de structure d'accueil en structure d'accueil. Dans le chapitre sur la mobilité, elles sont désignées par le terme "entité migrantes" qui ne sera pas repris ici pour éviter une confusion avec l'acception du terme entité au sens sécurité (élément actif) par opposition à un objet (élément passif).
- Le **réseau** reliant différentes structures d'accueil entre elles.

Cette architecture peut être soumise à différents types d'**attaques**: rejeu (attaque contre un système consistant à stocker et à réutiliser ultérieurement un message), impersonnation ou usurpation d'identité, interception

et/ou modification d'information, ces deux derniers types d'attaques mettant en péril les propriétés d'intégrité et de confidentialité. Ces attaques peuvent porter sur trois parties de l'architecture décrite précédemment.

(i) **Les communications entre deux structures d'accueil** doivent être sécurisées pour éviter la divulgation (confidentialité) ou la modification (intégrité) non autorisées d'information par un utilisateur malveillant. Les canaux de communication peuvent être sécurisés avec des protocoles dédiés utilisant des techniques cryptographiques comme SSL, ou des mécanimes d'authentification comme Kerberos. Ces techniques seront étudiées dans les sections 3.1 et 3.3. Dans le cas d'un système à objets mobiles, cet élément peut correspondre au cas où une unité d'exécution distante migre vers une structure d'accueil locale. Elle doit être authentifiée et certains droits d'accès doivent lui être accordés.

(ii) **Les communications entre une unité d'exécution et une structure d'accueil** doivent également être protégées. Cela peut signifier la protection des ressources d'une structure d'accueil contre des accès non-autorisés réalisés par une unité d'exécution, mais aussi la protection d'une unité d'exécution contre une structure d'accueil malveillante: celle-ci peut divulguer des informations sans autorisation de l'unité d'exécution, lui refuser l'accès à un service ou le surfacturer, etc. Ce cas ne sera pas traité ici, en faisant l'hypothèse d'une structure d'accueil sûre.

(iii) **Les échanges entre deux unités d'exécution** doivent également être protégés. Ce cas relève plus généralement de la sécurité des communications examinée en section 3.3.

La notion de **domaine de protection**, ou de domaine de sécurité est centrale dans ce modèle. On peut dire de manière générale, que dans ce modèle, la notion de domaine de sécurité correspond à la portée d'une structure d'accueil, en englobant toutes ses unités d'exécution. L'élément (i) correspond à de la **sécurité inter-domaines** (entre deux structures d'accueil), tandis que les éléments (ii) et (iii) relèvent de la **sécurité intra-domaines**. Cette remarque permet de faire le lien entre les autres chapitres du document qui s'articulent autour de la notion très générale de domaine, et l'analyse qui sera menée ici, plus axée sur les mécanismes de sécurité proprement dits.

La suite du chapitre présente la grille d'analyse retenue pour analyser les modèles et langages pour la mobilité de ce document sur un plan sécurité. Les sections suivantes tentent d'analyser les différentes possibilités de protection par type de fonction de sécurité mise en oeuvre: identification et authentification (section 3.1), contrôle d'accès (section 3.2), sécurité des communications (section 3.3), audit (section 3.4) et administration d'une politique de sécurité (section 3.5).

3.1 Identification et authentification

A la base de tout mécanisme d'authentification, on trouve un mécanisme d'**identification** i.e. chaque entité doit disposer d'une ou plusieurs identités difficilement falsifiables. Cette identité sera vérifiée au cours du processus d' **authentification**. On peut distinguer **authentification faible** comme la transmission d'un mot de passe et **authentification forte** où aucun secret n'est révélé au cours du processus d'authentification. L'authentification forte peut typiquement être réalisée à l'aide de systèmes de chiffrement asymétriques comme dans Agent Tcl. On peut également distinguer **authentification mutuelle** (deux processus sont impliqués) et **authentification via un tiers** (trois processus sont impliqués). C'est alors un serveur d'authentification, distinct du client et du serveur qui réalise la vérification de leurs identités. Un exemple d'authentification via un tiers est le système Kerberos où les deux entités qui cherchent à s'authentifier partagent une **clé de session** fournie par le serveur d'authentification. Cette clé de session peut également être utilisée pour chiffrer les communications.

Dans les modèles et langages étudiés, l'authentification, lorsqu'elle est prise en compte peut se faire à l'aide d'un serveur séparé comme dans Obliq (version sécurisée implémentée à l'aide de Secure Network Objects). Par contre dans un système comme Agent Tcl, l'authentification est mutuelle: la structure d'accueil joue le rôle de serveur d'authentification lors de l'authentification d'un agent par un serveur d'agents ou de l'authentification mutuelle de deux serveurs d'agents.

L'utilisation de calculs de processus orientés sécurité comme le spi-calcul permet typiquement de formaliser des protocoles d'authentification utilisés dans les systèmes réels. D'autres outils sont également disponibles comme la logique BAN de Burrows, Abadi, et Needham, ou des théories sur la confiance e.g. TCB (Trust Computing Base) de Lampson et al.

3.2 Contrôle d'accès

Les ressources dont on cherche à contrôler l'accès sont variées, typiquement un nom de canal (spi-calcul, seal calcul) ou un objet (Java, Obliq). Le contrôle d'accès peut s'effectuer également à différents niveaux de granularité: canal (spi-calcul), objet (Obliq), classe (Java), méthode (Java), groupe de classes (Java).

3.2.1 Politique d'accès aux ressources

La modélisation d'une politique d'accès et plus généralement d'une politique de sécurité trouve son origine dans la volonté de la séparer nettement des mécanismes qui l'implémentent.

Peu de modèles et langages à part Java disposent de primitives pour modéliser explicitement une politique d'accès.

3.2.2 Modèle d'autorisation

Un modèle communément adopté pour le contrôle d'accès est celui de la **matrice de contrôle d'accès** $M = (M_{ij})$ où i parcourt les entités, j parcourt les objets et M_{ij} représente le droit d'accès de l'entité i sur l'objet j e.g. lire, écrire, etc. Pour des systèmes réalistes, stocker l'ensemble de la matrice d'accès paraît impossible. Aussi, utilise-t-on deux représentations abrégées de cette matrice:

- **Les listes de contrôle d'accès ou ACL (Access Control Lists)** définissent pour un objet la liste des entités qui disposent de droits sur lui. C'est la représentation de la matrice d'accès colonne par colonne.
- **Les listes de capacités (Capability Lists)** définissent pour une entité la liste des objets sur lequel elle a des droits. C'est la représentation de la matrice d'accès ligne par ligne.

Le tableau 3.1 résume les avantages et les inconvénients de l'utilisation de ces deux représentations.

Opération	**ACL**	**Capacités**
Authentification	Facile en utilisant un serveur d'authentification externe.	Très facile car la capacité est détenue par l'entité. L'authentification peut donc se faire à l'intérieur de l'entité.
Recherche d'un droit d'accès	Facile car il suffit de parcourir la liste.	Difficile car il faut garder la trace de l'ensemble des capacités.
Révocations de droits d'accès	Facile car il suffit de modifier la liste.	Difficile car il faut invalider les capacités et garder une trace de l'ensemble des capacités invalidées.
Extension de droits d'accès	Facile car il suffit de modifier la liste.	Facile mais la diffusion des capacités peut affaiblir la sécurité du système.

TAB. 3.1 – *Différentes approches pour le contrôle d'accès.*

Certaines approches utilisent à la fois des capacités et des ACL pour contrôler l'accès: avec le concept de session, des ACL sont utilisées uniquement à la création d'une session. Le système fournit aux entités qui veulent accéder à des ressources pendant cette session des capacités pour que le contrôle d'accès se fasse de manière plus efficace. Les capacités sont détruites à la fin de la session. En général, les systèmes utilisant à la fois des capacités et des ACL nécessitent des adaptateurs pour faire interopérer les différents modèles d'autorisation, ce qui peut affaiblir la sécurité du système complet.

D'autres modèles que celui de la matrice d'accès peuvent être envisagés: dans un système à plusieurs niveaux de sécurité, la notion de **niveau d'autorisation** (security clearance) attachée à une entité, et celle de **niveau**

de sécurité attachée à un objet permettent de restreindre l'accès de l'entité aux objets pour lesquels le niveau de sécurité est égal à son niveau d'autorisation. Ces systèmes à bases de "labels" sont en général difficiles à administrer.

Dans tous ces modèles, on appelle **privilèges** les attributs de sécurité relatifs à des entités, et **attributs de contrôle** ceux relatifs à des objets.

Dans les langages et modèles étudiés, les approches décrites ci-dessus sont assez bien représentées: capacités (ambients, Pict, Java, Obliq, seal calcul), ACL (Obliq, Agent Tcl), niveaux de sécurité (Agent Tcl). Dans plusieurs calculs et langages (spi-calcul, seal calcul, ambients, Obliq), la portée lexicale permet de contrôler l'utilisation des noms de canaux vus comme capacités de communication ou plus généralement l'accès à des ressources. Dans d'autres cas (seal calcul), la restriction sur l'utilisation des noms de canaux sert à contrôler la migration.

Dans les différentes fiches, cette section tente de décrire comment les privilèges et attributs de contrôle détaillés précedemmment sont mis en oeuvre pour réaliser le contrôle d'accès.

3.2.3 Délégation de privilèges

Lorsqu'un client réalise une invocation, l'objet serveur peut à son tour réaliser des appels sur d'autres objets. Du point de vue du contrôle d'accès, le problème de la **délégation** consiste à savoir de quels privilèges va disposer un objet intermédiaire dans la chaîne d'invocations: ses privilèges propres ou ceux que lui auront délégués les objets à l'origine de l'appel. Ce problème ouvre la voie à des attaques comme l'usurpation d'identité qui doivent être combattues en utilisant conjointement des techniques de contrôle d'accès et d'authentification.

Dans les modèles étudiés, la délégation de droits d'accès n'est pas prise en compte. Ce problème est probablement encore trop ouvert pour être traité au niveau des modèles de programmation: seuls quelques systèmes comme des ORB (Object Request Broker) implémentant les spécifications CORBA fournissent quelques éléments de réponse au problème.

3.2.4 Domaine de sécurité

La notion de **domaine de sécurité ou de protection** permet de grouper un ensemble d'objets auquels on accorde les mêmes attributs de contrôle, afin de réduire la taille de la matrice de contrôle d'accès. Elle est à distinguer d'un autre type de domaine, celui de **groupe d'utilisateurs**, qui rassemble un ensemble d'entités avec les mêmes privilèges.

Plusieurs calculs ou langages étudiés modélisent un domaine de protection par la notion de **portée** (ambients, spi-calcul, seal calcul, Kali Scheme, Obliq), l'opérateur de restriction permettant de restreindre les noms connus par un processus, donc de garantir des propriétés d'intégrité et de confidentialité puisque ces noms ne peuvent sortir de la frontière définie par la portée à moins d'utiliser une règle d'extension de portée. Une telle règle peut modéliser le passage d'un domaine de protection à un autre (spi-calcul, ambients, seal calcul). Des restrictions peuvent être imposées sur l'utilisation de cette règle (seal calcul). Dans les langages et modèles étudiés, la gestion des domaines reste implicite (Java). En général, à un domaine de protection est attachée une polique de sécurité, mais aucun langage à part Java ne prend ce fait en compte explicitement. Certains langages disposent également de la notion de **domaine de confiance** (Agent Tcl).

3.3 Sécurisation des communications

Les communications sont sécurisées, principalement en utilisant des techniques cryptographiques. On dispose de deux grandes méthodes pour garantir la confidentialité des messages: les algorithmes de chiffrement symétriques (e.g. DES, IDEA) ou asymétriques (e.g. RSA). L'intégrité des messages est garantie à l'aide d'algorithmes de signature qui utilisent des fonctions de condensation (e.g. MD5)[1]. Les systèmes évolués de sécurité offrent des mécanismes de gestion de certificats[2] pour assurer la non-répudiation[3] des messages.

La plupart des calculs modélisent la confidentialité et l'intégrité des messages à l'aide de la portée lexicale. Seuls les calculs spécifiquement orientés sécurité (spi-calcul, sjoin) vont au-delà et intègrent dans leur syntaxe des primitives pour le chiffrement ou proposent des extensions allant dans ce sens (Kali Scheme, seal calcul). Les

1. Une **fonction de condensation** est une fonction à sens unique qui produit un "condensé" du message, par exemple un hash.
2. Un **certificat** est un document écrit ou sous forme électronique qui émane d'une autorité reconnue et atteste d'un fait, d'un droit e.g. certificat de réception ou d'envoi d'un message.
3. La **non-répudiation** est la caractéristique de tout message (ou bloc d'informations) dont l'émetteur et le récepteur ne peuvent renier avoir émis ou reçu.

langages prenant vraiment en compte la sécurité comme Java ou Agent Tcl disposent de fonctions de chiffrement symétriques ou asymétriques, ainsi que des algorithmes de signature. Seul Java semble offrir des outils de gestion de certificats.

3.4 Audit

Les outils d'**audit** permettent de détecter les tentatives de violation de la sécurité d'un système en enregistrant certains événements. La politique d'audit va déterminer quels sont les événements relatifs à la sécurité à retenir afin d'éviter une explosion du nombre d'enregistrements.

Dans les modèles et langages étudiés, l'audit n'est pas pris en compte. Cette fonction doit être réservée à des systèmes ayant des besoins de protection importants. Un ORB CORBA permet par exemple de spécifier une politique d'audit, mais on sort ici du cadre des langages fixé dans ce document.

3.5 Administration d'une politique de sécurité

La gestion d'un domaine de sécurité doit comporter:

- **La gestion du domaine lui-même:** création, suppression et positionnement d'un domaine dans une hiérarchie de domaines.
- **La gestion des membres du domaine:** déplacement d'un objet d'un domaine à un autre.
- **La gestion de la politique associée à ce domaine:** e.g. protection par défaut des messages, politique de délégation, d'audit, de non-répudiation, de contrôle d'accès.

Dans les langages et systèmes étudiés, seuls Java et Agent Tcl prennent en compte explicitement le dernier élément. La définition d'une politique de sécurité globale pour un domaine par fédération des politiques de ses sous-domaines n'est pas abordée. Les deux premiers éléments restent implicites au niveau des modèles et des langages. Il faut se tourner vers des implémentations comme un ORB CORBA pour qu'ils soient pris en compte explicitement au niveau des Common Facilities.

Deuxième partie

Langages et Modèles

Chapitre 1

Agent Tcl

1.1 Références bibliographiques

R.S Gray, Agent Tcl: alpha release 1.1 (1995), disponible à l'URL

http://www.cs.dartmouth.edu/~agent

1.2 Présentation générale

Agent Tcl est une extension du langage de scripts Tcl, qui permet la programmation de systèmes d'agents migrants et communicants au cours de leurs calculs.

1.3 Critère "distribution"

1.3.1 Entités de base pour la distribution

Entités de base

Agent Tcl ajoute au langage Tcl un jeu d'instructions qui permettent la création et la migration d'agents, qui sont ultimement des scripts Tcl. Les entités de base, mis à part ces scripts sont les agents et les serveurs, qui sont nommés de façon unique.

Un agent est normalement un script débutant par une instruction `agent_begin` (avec un paramètre optionnel spécifiant la machine sur laquelle l'agent s'exécute), qui crée un nom (nouveau), et finissant par `agent_end`. On peut spécifier un nom symbolique par la commande `agent_name`.

Manifestations de domaines

Un "domaine" dans Agent Tcl est un "serveur" installé sur une machine, chargé de l'exécution des agents. Le serveur comporte un interprète de scripts, et gère la migration des agents. Ce que l'on connait d'un serveur au niveau du langage est son nom.

Noms/références

Un serveur est identifié par l'adresse Internet de la machine hôte. Le nom d'un agent est composé du nom de l'interprète qui l'exécute, et d'un nom propre unique.

1.3.2 Concept de domaine

Topologie de l'espace des domaines

Il est plat, chaque serveur étant lié à une machine. Un serveur n'accepte d'agent migrant que venant d'une machine faisant partie d'une liste connue à l'avance.

Sémantique des domaines

Un domaine pour Agent Tcl est une unité d'exécution. Etant lié à une machine, un serveur est aussi une unité de défaillance, bien que rien ne permette dans le langage de manipuler cet aspect.

1.3.3 Communication locale

La communication a lieu entre agents. Il n'y a pas de différence entre communication locale et distante (cette dernière étant le mode "par défaut").

Désignation locale

Le nom d'un agent pour la communication est constitué de l'adresse (Internet) du serveur qui exécute cet agent et du nom (unique) de l'agent lui même.

Politique de communication locale

Il y a deux façons de communiquer entre agents: par messages et par "rencontre". Pour envoyer un message on utilise l'instruction `agent_send` qui a deux arguments, le premier étant le [nom du] destinataire, qui est un agent, le second étant le contenu du message, qui est du type "string". L'envoi de message est bloquant, mais il y a un "timeout" par défaut (toutes les instructions d'Agent Tcl ont une version "temporisée"). La communication est monadique et point à point.

La "rencontre" (meeting) est l'établissement d'une communication directe (par lecture/écriture sur des "sockets") entre deux agents. On l'obtient par synchronisation de commandes `agent_meet`, qui désigne directement le partenaire, et `agent_accept` (il y a d'autres formes d'acceptation ou de rejet d'une rencontre). La sémantique est la même que pour la communication par message.

Sémantique de la réception

La primitive `agent_receive` a une forme bloquante et une forme non-bloquante (dans cette forme on ne sait pas ce qu'il advient de la variable à laquelle est supposé être lié le contenu du message). On a toujours unicité du récepteur puisque celui-ci est un agent (qui est l'agent qui exécute la commande `agent_receive`, qui n'est pas désigné explicitement).

1.3.4 Communication distante

Désignation distante

Le nom du destinataire d'un message comporte toujours sa localisation.

Politique de communication distante

Il semble que si l'agent destinataire d'un message a migré, l'émetteur est simplement bloqué (la communication est synchrone).

1.4 Critère "mobilité"

1.4.1 Nature des entités migrantes

Les entités migrantes sont les agents.

1.4.2 Expression de la migration

Il y a trois formes: l'instruction `agent_submit`, qui a pour paramètre le nom d'un serveur, provoque la création d'un nouvel agent à destination. Le corps de cet agent est un script, argument de cette commande. On doit spécifier aussi la liste des variables et des procédures définies dans l'agent "père" et qui sont utilisées par l'agent créé à distance. A la terminaison, la valeur du "fils" est automatiquement envoyée à l'agent créateur.

L'instruction `agent_jump`, qui a pour paramètre le nom d'un serveur, provoque la migration de l'agent qui l'exécute. Cette forme de migration est bloquante vis à vis de l'entité migrante, puisque l'exécution de l'agent

reprend à partir de l'instruction qui suit la migration. L'instruction `agent_fork`, qui a pour paramètre le nom d'un serveur, crée une copie de l'agent qui l'exécute. Le serveur qui reçoit ce "clone" lui attribue une nouvelle identité.

1.4.3 Mobilité en cours d'exécution

L'instruction `agent_submit` est une forme de "code shipping", tandis que les deux autres instructions de migration autorisent la mobilité au cours de l'exécution d'un agent. On peut parler de mobilité de continuation; en particulier, un agent ne peut être à la fois capable de recevoir ou émettre un message et sujet à migration (un agent est toujours séquentiel).

1.4.4 Migration d'un état associé

L'exécution d'une instruction `agent_submit` provoque la copie de la valeur des variables et des procédures qui sont spécifiées en argument de cette instruction (il n'y a pas d'indication sur ce que peut être cette valeur – clôture?).

Les instructions `agent_jump` et `agent_fork` provoquent la copie de l'état de l'agent au moment de leur exécution, sans que la notion d'état soit autrement précisée (que se passe-t-il si une portion de cet état est partagée?).

Toutes les formes de migration comportent un déplacement et une mise à jour de la table associée à l'agent qui contient tous les traits relatif à son identification.

1.4.5 Entretien de références distantes

Bien qu'il y ait une forme de désignation globale des agents (leur nom comportant leur localisation), il n'y a pas de "suivi" d'un agent migrant, donc pas de notion de référence distante à un agent en tant que tel (i.e. indépendamment de sa localisation).

1.5 Critère "sécurité"

Le modèle de sécurité d'Agent Tcl vise à protéger une machine contre un agent malveillant ou à protéger les agents entre eux. Un serveur accepte un agent qui parvient sur un site local, l'authentifie et transmet l'agent authentifié à l'interprète approprié. Une fois l'authentification établie, Agent Tcl utilise des agents gestionnaires de ressources (AGR) pour contrôler l'accès aux ressources critiques et le modèle de sécurité de Safe Tcl pour assurer une exécution sûre de l'agent compte tenu des restrictions imposées par les AGR.

1.5.1 Identification et authentification

Chaque agent et chaque serveur disposent d'une identité. L'authentification est basée sur le protocole PGP (Pretty Good Privacy)[1]. L'authentification se fait en utilisant des techniques cryptographiques, un agent prouvant son identité par connaissance d'une clé.

On a plusieurs situations possibles:

- Un agent s'enregistre avec un serveur: l'agent utilise sa clé secrète pour signer la requête d'enregistrement, la chiffre avec la clé publique du serveur, puis la lui transmet.
- Un agent migre: sa requête est signée avec la clé privée du serveur, et chiffrée avec la clé publique du destinataire. Ceci requiert un degré de confiance élévé entre les différents serveurs car le message est signé à l'aide de la clé du serveur. L'identité de l'agent est transmise lors de la migration. Si l'agent est authentifié, le serveur retient l'identité de l'agent et de son serveur de départ, et le niveau de confiance qu'il peut accorder à ce serveur.
- Un agent envoie un message à un autre agent situé sur une machine différente ou crée un agent à distance: la démarche est identique.

Le système d'authentification d'Agent Tcl a deux faiblesses:

- On ne dispose pas de service de distribution de clés PGP.

1. Un message est chiffré à l'aide de l'algorithme à clé secrète IDEA. Cette clé est générée au hasard et chiffrée à l'aide de la clé publique du destinataire en utilisant l'algorithme RSA. La clé IDEA et le message chiffré sont ensuite envoyés au destinataire. Une option est disponible pour signer le message en utilisant l'algorithme MD5.

– Le système est vulnérable à des attaques de rejeu.

1.5.2 Contrôle d'accès

On distingue deux types de ressources:

– Les **ressources primitives** sont directement accessibles à travers des primitives du langage. Des ressources primitives typiques sont l'écran, le système de fichiers, l'horloge ou le réseau.

– Les **ressources indirectes** ne peuvent être accédées que par l'intermédiaire d'un autre agent.

Politique d'accès aux ressources

Elle n'est pas explicitée. Elle est contenue dans les listes d'accès contenues dans l'interpréteur sûr (cf. ci-dessous).

Modèle d'autorisation

– Pour les ressources indirectes, l'agent A gérant la ressource se charge d'en contrôler l'accès, en maintenant des listes d'agents autorisés à utiliser cette ressource. A tient compte des informations suivantes pour autoriser l'accès: identité de l'agent B désirant utiliser la ressource et de son serveur rattaché, caractère authentifié de B et de son serveur, niveau de confiance que A peut placer dans le serveur de B.

– Pour les ressources primitives, le contrôle d'accès s'effectue par l'utilisation de Safe Tcl et d'AGR. Pour chaque agent, Safe Tcl fournit un interpréteur sûr et un interpréteur non-sûr où les commandes "dangereuses" sont renvoyées vers l'interpréteur sûr qui autorisera ou non leur exécution en consultant des listes d'accès du type (nom de resssource, nombre autorisé d'instances de cette ressources). Ces listes sont détenues par les AGR. Un agent peut demander explicitement à un AGR l'autorisation d'utiliser une ressource primitive avec le mot clé `require`: cela se traduit par un renvoi de cette commande vers l'interpréteur sûr. Le contrôle est effectué implicitement sinon.

Les versions futures d'Agent Tcl supporteront la protection d'un groupe de machines contre un agent malveillant, pour éviter que l'agent abuse des ressources globales du groupe sans abuser des ressources de chaque machine. Ces mesures seraient fondées sur des mécanismes de type porte-monnaie électronique, chaque agent disposant d'un certain nombre de crédits qu'il dépense à chaque fois qu'il utilise des ressources.

Domaine de sécurité

Lors des relations agent-serveur, le serveur consitue un domaine de confiance: un serveur fera confiance à un agent extérieur s'il fait confiance au serveur auquel cet agent appartient. Lors des relations agent-agent, c'est l'agent qui consitue l'unité de confiance.

Un serveur consitue également un domaine de protection, un interpréteur sûr étant associé à un serveur, et contenant les droits pour les ressources primitives. Les AGR font aussi partie de ce domaine de protection.

1.5.3 Sécurisation des communications

On a plusieurs cas possibles:

– Lors de l'enregistrement entre un agent et un serveur, toutes leurs communications ultérieures sont chiffrées à l'aide d'une clé de session e.g. la clé secrète IDEA.

– Lors de la création d'un agent à distance, son script est signé et chiffré avant d'être envoyé sur le serveur distant.

– Lors de la migration d'un agent, son image est signée et chiffrée avant d'être envoyée sur le site distant.

1.5.4 Audit

Fonction envisagée dans les versions futures d'Agent Tcl.

1.5.5 Administration d'une politique de sécurité

On dispose d'un agent `console` qui garde la trace d'agents s'exécutant sur une machine, et peut y interdire la migration de nouveaux agents, ou détruire des agents en cours d'exécution.

Pour l'instant, les listes d'accès contenues dans l'interpréteur sûr ne sont pas modifiables. Dans des versions futures d'Agent Tcl, on pourra définir une politique de sécurité paramétrable pour un serveur ou pour un groupe de serveurs.

Chapitre 2

Ambients

2.1 Références bibliographiques

L.Cardelli, A.Gordon, Mobile Ambients, FoSSaCS'98, LNCS 1378 (1998) 140-155.

2.2 Présentation générale

Un "ambient" est un "enclos", où ont lieu des traitements. Le calcul des ambients s'inspire de l'idée de hiérarchie de domaines administratifs, et de hiérarchie d'autorisations mise en oeuvre entre ces domaines. Il capture de manière primitive les concepts de domaines, de mobilité, et d'autorisation associée à la mobilité.

Dans les ambients calculer signifie migrer: les primitives de mobilité sont suffisantes pour conférer au calcul un caractère universel (on peut coder les machines de Turing). La communication est ajoutée par commodité.

2.3 Critère "distribution"

2.3.1 Entités de base pour la distribution

Entités de base

Les entités de base sont les "domaines" – appelés "*ambients*" – les *processus* et les *capacités* – à migrer au travers d'une frontière ou à dissoudre une frontière. Les ambients contiennent des processus; ces derniers sont des compositions parallèles d'actions, exhibant une capacité, et d'ambients.

Manifestations de domaines

Le concept de domaine se réalise sous la forme d'*ambients*. Un ambient est un conteneur (local fermé) abritant une composition parallèle de processus et d'ambients, et délimité par une frontière.

Noms/références

Le calcul des ambients manipule des noms (d'ambients), régis par une règle de portée syntaxique. Un nom d'ambient peut désigner plusieurs ambients à l'intérieur de sa portée.

2.3.2 Concept de domaine

Topologie de l'espace des domaines

L'espace des ambients est hiérarchique. L'éther entre les ambients n'est pas formalisé.

Sémantique des domaines

Un ambient est une unité de communication et de migration. Un ambient peut migrer de manière atomique, et il emporte avec lui tous ses ambients fils et les processus qu'il contient. Le nom de l'ambient constitue un "mot de passe" qu'il faut connaitre pour le quitter, s'y rendre ou le dissoudre.

Observabilité/contrôlabilité des domaines

La frontière d'un ambient peut être ouverte, de manière objective par la capacité *open* (une forme subjective "*acid*" est "programmable"). Dans une faible mesure, on peut considérer que les processus abrités par un ambient (et qui détiennent la capacité de le faire migrer par exemple) en assurent le contrôle.

On peut aussi "activer" un ambient en cours de calcul, soit en exécutant des capacités qui le libèrent, soit à la suite d'une communication.

2.3.3 Communication locale

Désignation locale

La désignation locale d'un canal est sans objet dans le calcul des ambients (en effet, la communication est anonyme).

Politique de communication locale

La communication locale est asynchrone et anonyme (pas de canal cible désigné), et univoque. Les arguments de la communication sont des noms, des capacités ou des séquences d'arguments.

2.3.4 Communication distante

Elle n'existe pas: la communication ne peut avoir lieu qu'à l'intérieur d'un ambient donné.

2.4 Critère "mobilité"

2.4.1 Nature des entités migrantes

Ce sont les ambients qui migrent. On peut "programmer" une forme de migration des processus par dissolution de l'ambient qui les contient (i.e. en utilisant la primitive *open*).

2.4.2 Expression de la migration

Il y a deux instructions de migration – aussi appelées "capacités" –, l'une permettant à un ambient de sortir de l'ambient qui le contient (devenant ainsi un frère de son père), l'autre permettant à l'inverse d'entrer chez un voisin (devenant le fils d'un frère). Ces deux instructions sont subjectives et asynchrones par rapport au contenu de l'ambient dont elles provoquent la migration, mais elles contrôlent un processus "continuation" qui sera exécuté une fois la migration effectuée. On ne peut donc "contrôler" le contenu de ce qui migre.

Une instruction de migration désigne directement, par son nom, la "cible" de la migration (l'ambient dont on veut sortir ou dans lequel on veut entrer), mais l'ambient concerné par la migration reste implicite: c'est celui qui contient l'instruction.

2.4.3 Mobilité en cours d'exécution

Elle se manifeste de deux façons: d'une part un ambient peut migrer "en cours d'exécution" (si l'on considère que l'exécution d'un ambient est celle de son contenu), et d'autre part, les capacités à migrer bloquant une continuation, on peut les faire se succéder dans le temps (par exemple pour se déplacer dans la hiérarchie des ambients).

2.4.4 Migration d'un état associé

Il n'y a pas de notion d'état dans le calcul des ambients. Il n'y a pas de déplacement d'un "contexte" d'un ambient au cours de la migration, si l'on entend par contexte l'ensemble des ambients dont le migrant connait le nom.

2.4.5 Entretien de références distantes

On peut dire qu'un ambient conserve la connaissance des noms d'ambients sur lesquels il possède une référence, mais on ne peut pas considérer qu'il s'agit de références distantes. En effet ces noms désignent (éventuellement) des ambients du contexte d'arrivée du migrant, et non ceux auxquelles ils faisaient référence avant le déplacement.

2.5 Critère "sécurité"

La sécurité dans les ambients se limite aux droits d'accès à un ambient, et est capturée par la portée syntaxique des noms d'ambients (restriction d'influence des noms).

Chapitre 3

Le π-calcul Distribué (Dπ)

3.1 Références bibliographiques

M. Hennessy, J. Riely, Resource access control in systems of mobile agents, Techn. Report 2/98, School of Cognitive and Computer Sciences, University of Sussex (1998)

R. Amadio, G. Boudol, C. Lhoussaine, The receptive distributed π-calculus, Workshop on Mobile Object Systems, Lisboa, June 1999.

3.2 Présentation générale

Le Dπ-calcul de Hennessy et Riely est un π-calcul synchrone, enrichi de primitives pour la migration (notion de localité, instruction de migration, noms structurés). Sa version "réceptive" est asynchrone, avec une simplification en ce qui concerne les noms structurés. Sa caractéristique, relativement à π_{1l}, est le choix d'un mode de communication purement locale, avec une gestion explicite de la migration des messages vers des récepteurs distants.

3.3 Critère "distribution"

3.3.1 Entités de base pour la distribution

Entités de base

Les entités de base sont les processus (aussi appelés "threads") et les configurations, aussi appelées systèmes. Une configuration est un système de processus localisés (dans des "domaines") en parallèle. Dans la deuxième version, la syntaxe est un peu plus simple, un processus migrant n'étant pas distingué d'un processus localisé.

Manifestations de domaines

Un domaine se manifeste sous la forme d'une construction particulière du calcul, qui est une paire constituée d'un nom et d'un processus. Il y a un seul domaine de nom donné, même s'il peut se présenter sous la forme de différentes parties (de même nom) en parallèle.

Noms/références

On a deux sortes de noms dans Dπ, les noms de canaux et les noms de localités, à quoi la seconde version ajoute des noms désignant des valeurs (sur lesquelles on peut faire un test d'égalité). De plus, on a des noms composés, qui consistent essentiellement en une paire d'un nom de localité et d'un nom de canal (dont le récepteur est présent à la localité indiquée). Ces noms sont utilisés seulement en argument, mais non pour désigner directement un canal par exemple. Tous ces noms obéissent à la discipline de la portée lexicale, la portée pouvant être répartie sur plusieurs localités.

3.3.2 Concept de domaine

Topologie de l'espace des domaines

L'espace des localités est plat, non hiérarchique. Chaque localité est supposée directement accessible depuis les autres.

Sémantique des domaines

Une localité de Dπ est une unité d'exécution et de communication: à l'intérieur d'un "domaine" les processus s'exécutent et peuvent communiquer entre eux, par envoi de messages comme dans le π-calcul de base.

3.3.3 Communication locale

Désignation locale

La désignation est directe: on utilise simplement le nom du canal de communication.

Politique de communication locale

C'est celle du π-calcul (synchrone ou asynchrone), polyadique et univoque (un seul récepteur est concerné par un envoi de message).

Sémantique de la réception

Dans la première version, il n'y a pas de contrainte – c'est la réception du π-calcul. Dans la seconde version, la réception est "statique", c'est à dire qu'on ne peut créer un récepteur pour un nom qu'on a reçu; un tel nom ne peut être utilisé que comme argument ou destinataire de message. De plus, on a localement la propriété de récepteur unique (au plus un récepteur par canal dans chaque localité) et de réceptivité, c'est à dire qu'un récepteur est toujours disponible pour traiter des messages.

3.3.4 Communication distante

Il n'y a pas de communication distante: un message pour lequel le récepteur correspondant se trouve dans un domaine distant doit faire l'objet d'une migration explicite vers son destinataire. Dans la seconde version, une analyse statique permet d'assurer qu'un message ne peut être envoyé vers une destination où il ne trouverait pas de récepteur.

3.4 Critère "mobilité"

3.4.1 Nature des entités migrantes

Les entités migrantes sont les processus du calcul; les localités ne peuvent migrer. Dans le calcul d'Hennessy et Riely il y a des contraintes de type à vérifier, qui imposent qu'un processus migrant utilise les noms de façon cohérente avec ce qui est prescrit par le type de la localité de destination.

Dans la seconde version, on a la même contrainte de typage, mais de plus il faut assurer l'unicité du récepteur, et la réceptivité, ce qui conduit à interdire de fait la migration en cours d'exécution d'un processus qui contiendrait un récepteur sur un canal public. Ainsi, les diverses localisations possibles de recépteurs sur un nom de canal donné sont statiquement déterminées. Il est cependant toujours possible d'installer dynamiquement à distance un récepteur sur un canal nouveau.

3.4.2 Expression de la migration

La migration s'exprime sous la forme d'une instruction (processus élémentaire) $[a :: P]$ par laquelle le processus P est envoyé à la localité a pour s'y exécuter. C'est donc une mobilité objective, asynchrone (mais il est facile d'en programmer une version synchrone). Comme dans π_{1l}, le le processus P est passif, et ne commence son exécution qu'une fois parvenu à destination. Contrairement au cas de π_{1l}, la migration est ici une primitive nécessaire, ne serait-ce par exemple que pour expédier un message vers un récepteur distant, puisqu'il n'y a pas de communication globale.

3.4.3 Mobilité en cours d'exécution

La mobilité au cours des calculs s'exprime par l'imbrication des instructions de migration.

3.4.4 Migration d'un état associé

Il n'y a pas de migration d'état, au sens où le contexte d'un processus migrant est toujours fixe – sauf en ce qui concerne les définitions récursives de processus, qui sont substituées aux appels dans le migrant. Ces définitions récursives sont associées à un nom de récepteur unique, et possèdent des paramètres, qu'on peut considérer comme des variables d'état. Par conséquent, lorsque la migration concerne un récepteur, l'entité migrante comporte une information sur l'état courant de "l'objet" désigné par le nom du canal.

3.4.5 Entretien de références distantes

Le processus migrant conserve les liens qu'il possède, puisque ceux-ci sont matérialisés par un nom de canal. Toutefois, ces liens distants ne peuvent être utilisés directement pour la communication. Le système d'analyse statique du Dπ réceptif impose au "programmeur" d'installer un délégué pour tout récepteur qu'un processus migrant peut recréer (sous un nouveau nom) dans sa localité de destination. De même, si le processus migrant contient un message pour un récepteur qui ne se trouve pas à sa localité de destination, le "programmeur" se voit contraint par le système d'analyse statique d'acheminer explicitement ces messages vers une destination correcte.

Chapitre 4

Java

4.1 Références bibliographiques

J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison-Wesley, 1996.
T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1999.
Sun Microsystems, Java Remote Method Invocation Specification, October 1998.
L. Gong, Inside Java 2 Platform Security: Architecture, API Design, and Implementation, Addison-Wesley, 1999.
Page de référence pour la documentation sur la plate-forme Java:

http://java.sun.com/docs/index.html

4.2 Présentation générale

Java est un langage de programmation de Sun Microsystems qui s'est développé à partir d'un langage nommé **Oak** initialement prévu pour programmer des périphériques comme des PDA (Personal Digital Assistant) ou des STB (Set Top Box). Le langage Oak était efficace, robuste, indépendant d'une architecture et orienté-objets.

Java dispose de caractéristiques similaires à Oak: ce langage orienté-objets est portable (le langage est semi-compilé ce qui offre un certain degré de portabilité par l'intermédiaire du byte-code), dynamique (les classes sont chargées à la demande), efficace et prend en compte la sûreté et la sécurité. La politique marketing intensive dont il a fait l'objet l'a imposé comme langage de l'Internet.

A partir de la version 1.1 de Java, Sun fournit un mécanisme d'invocation de méthode à distance appelé **Java RMI (Remote Method Invocation)** pour communiquer entre différents espaces d'adressage. Un stub va gérer l'appel sur un objet distant, de telle sorte qu'il soit vu par le client comme un appel local. Le but est de pouvoir écrire des applications réparties de manière efficace, tout en préservant les caractéristiques de sûreté et de sécurité de Java.

Java RMI permet de s'interfacer avec du code existant ou des bases de données en utilisant JNI (Java Native Interface) et JDBC (Java DataBase Connectivity). Java RMI offre une forme limitée de mobilité de code avec le chargement dynamique de stubs.

4.3 Critère "distribution"

4.3.1 Entités de base pour la distribution

Entités de base

Dans Java RMI, on dispose des notions suivantes utiles pour la répartition:

- Un **objet distant** (appelé aussi **objet servant**) est un objet dont les méthodes peuvent être invoquées à partir d'une autre machine virtuelle. Il implémente une ou plusieurs **interfaces distantes**. Les clients n'interagissent avec les objets distants qu'au moyen de leurs interfaces distantes. Ils ne sont pas en contact direct avec l'implémentation de l'objet.
- Un **serveur** regroupe plusieurs objets servants. En général, il implémente un service. Un site physique abrite plusieurs serveurs.

Manifestations de domaines

Au niveau système, on dispose d'une forme de domaine de nommage avec la notion de `ClassLoader`. Une classe est référencée à l'intérieur de la machine virtuelle à la fois par son nom et par le nom du `ClassLoader` qui l'a chargée. Le `ClassLoader` joue également le rôle de domaine de protection. Au niveau applicatif, on dispose d'une autre forme de domaine de nommage, les classes étant organisées en hiérarchies de packages. A partir de la version 1.2 du JDK, on dispose de la notion explicite de domaine de protection. Un serveur i.e. une machine virtuelle joue également le rôle de domaine d'exécution.

Noms/références

Java utilise comme notion de référence en contexte réparti celle de **stub** qui sera transmis d'un espace d'adressage à un autre à la place de l'objet référencé. La portée d'une référence correspond à sa portée lexicale.

4.3.2 Concept de domaine

Cf. section "manifestation de domaines". Les `ClassLoader` peuvent être organisés de manière hiérarchique. Il en est de même pour les packages. Les serveurs ont par contre une structure plate.

4.3.3 Communication locale

Désignation locale

A l'intérieur d'un package, elle se fait directement par une référence. Le nom complet (chemin dans l'arbre des packages) doit être utilisé si l'objet référencé appartient à un autre package.

Politique de communication locale

La communication est synchrone puisque Java est fondé sur l'invocation de méthode à distance à la RPC (Remote Procedure Call). La persistance est prise en compte en utilisant des références "intelligentes" qui contiennent à la fois une référence persistante et une référence non-persistante[1].

Sémantique de la réception

La réception est unique (une référence désigne un seul objet). Lors d'une invocation, l'appel de méthode sera reçu au plus une fois par l'objet référencé. Le contrôle de concurrence est géré à l'aide d'un verrou associé à l'objet référencé lorsqu'il contient des méthodes déclarées `synchronized`.[2]

4.3.4 Communication distante

Désignation distante

Par rapport à un objet simple, un objet distant doit implémenter l'interface `java.rmi.Remote`. Une référence peut être créée dans Java RMI à l'aide de la méthode `exportObject` des classes `UnicastRemoteObject` ou `Activatable`: cette méthode qui prend en entrée un objet distant va renvoyer une référence sous forme d'un stub. Dès lors, l'objet va écouter sur un port paramétrable les requêtes venant de clients éventuels.

Dans Java RMI, le service de nommage porte le nom de `rmiregistry`. Il crée une association entre une référence et un nom symbolique. Les noms sont des URL au format `//host :port/name` où `(host,port)` identifie le site sur lequel tourne le registry, et `name` est le nom sous lequel est identifié l'objet distant. Un registry peut être partagé par plusieurs serveurs sur un site. Chaque processus serveur peut aussi avoir son propre registry. Il est également possible d'utiliser un service de nommage où les noms ne sont pas nécessairement des URL.

1. La **référence non-persistante dite "vive"** sera utilisée pour contacter l'objet une fois activé, les appels étant transmis de la référence "intelligente" vers la référence non-persistante. Elle contient `null` si l'objet n'est pas activé. La **référence persistante** permet d'accéder à des informations utilisées lors du protocole d'activation afin d'obtenir une référence vive sur l'objet.

2. Lorsque cette option est appliquée à une méthode de classe, le verrou est associé au niveau de la classe et non plus au niveau d'une instance. Cette option peut également être utilisée pour définir des sections critiques pour un bloc d'instructions.

Politique de communication distante

Il y a cohérence entre la sémantique des références locales et distantes. Les arguments d'un appel de méthode distante doivent implémenter l'interface de sérialisation pour pouvoir être transmis sur le réseau. Les arguments non-distants d'une méthode sont passés par copie plutôt que par référence. Les arguments qui sont des objets distants sont passés par référence: i.e. c'est le stub correspondant à cet objet qui est transmis.

Routage logique

Le mécanisme d'invocation à distance de Java s'appuie sur la sérialisation qui permet de transmettre le byte-code d'un objet d'un espace d'adressage à un autre. La sérialisation des objets s'effectue à l'aide d'un mécanisme de streams, chaque classe étant annotée avec un URL qui représente l'endroit à partir duquel elle peut être chargée dynamiquement. Pour assurer la transparence à la répartition, le modèle de liaison de Java est à base de stubs/skeletons[3].

Java RMI consiste en une architecture en trois couches: une couche de stubs/skeletons indépendante de la sémantique d'une référence, une couche de gestion des références implémentant une sémantique particulière, et une couche de transport gérant les connections et la transmission des données.

4.4 Critère "mobilité"

4.4.1 Nature des entités migrantes

Java permet la mobilité des noms, les références d'objet pouvant être transmises sans contrainte entre différents espaces d'adressage. Il supporte aussi une forme faible de mobilité de code, où seul du code exécutable (byte-code) migre entre différentes machines virtuelles.

4.4.2 Expression de la migration

Java obéit au modèle de chargement dynamique de code: les classes sérialisées sont annotées avec des éléments de localisation pour qu'elles puissent être chargées dynamiquement par le récepteur. Ces informations peuvent prendre la forme d'un URL, mais dans ce cas, un serveur HTTP devra s'exécuter sur le site où réside la classe à télécharger.

Lors de la désérialisation, la résolution de classe s'effectue tout d'abord dans le contexte local du récepteur. Si cette résolution échoue, le récepteur peut tenter de charger dynamiquement des données sur la classe recherchée ou sur une ses instances. C'est en particulier le cas pour le chargement dynamique des stubs correspondant à un objet ou une classe distante.

La migration s'effectue par réplication: dans Java RMI, un objet distant est dupliquable s'il implémente l'interface `java.lang.Cloneable`. Il sera dupliqué en appelant la méthode `clone` de `java.lang.Object`.

4.4.3 Mobilité en cours d'exécution

La migration de code s'effectue au moyen de la sérialisation dans Java RMI, ou du chargement de byte-code pour les applets. Il n'y a donc pas de "mobilité en cours d'exécution", l'exécution du code étant gelée avant la migration.

4.4.4 Migration d'un état associé

Pour des applets Java, on est dans un modèle sans état. C'est le byte-code d'une classe qui est transmis. Si cette classe contient des références vers d'autres classes, celles-ci seront chargées à leur tour par le `ClassLoader`.

Dans Java RMI, l'implémentation d'un objet distant reste dans la machine virtuelle qui l'a créé et ne peut pas la quitter. C'est le stub pour cet objet distant qui sera transmis sur le réseau, et éventuellement téléchargé dynamiquement.

3. Les stubs contiennent l'ensemble des méthodes définies comme distantes dans l'interface de l'objet, mais ne contiennent pas les méthodes non-distantes. A partir de la version 1.2 de Java, l'API réflexive de Java est utilisée pour réifier une requête dans un objet de type `Method`. C'est la méthode `invoke` qui permet de réaliser l'invocation de méthode proprement dite. Dans la version 1.1 de Java, les skeletons récupèrent un objet représentant l'invocation et le dirigent vers un dispatcher. Ces skeletons sont supprimés dans Java 1.2, grâce à l'utilisation de la réflexion.

4.4.5 Entretien de références distantes

Les objets sont par défaut non-mobiles, seules des références étant transmises. Si une classe téléchargée référence des ressources externes, celles-ci sont répliquées sur le site d'origine.

4.5 Critère "sécurité"

On dispose en Java de plusieurs modèles de sécurité:

- Le modèle de sécurité du JDK 1.0 est celui de la "**sandbox**", forme de domaine de protection: on fait confiance au code local qui dispose d'un accès complet aux ressources système. Le code téléchargé n'a accès qu'aux ressources limitées contenues dans la sandbox. Le `Security Manager` contrôle les accès aux ressources [4].
- Avec le JDK 1.1, le modèle est étendu avec la notion d'**applet signée**: une applet signée peut accéder à l'ensemble des ressources comme le ferait du code local si sa signature est reconnue . Dans le cas contraire, elle doit s'exécuter à l'intérieur de la sandbox.
- Avec le JDK 1.2, l'ensemble du code peut être soumis à une **politique de sécurité** définie par un ensemble de droits d'accès à différentes ressources. Le code est organisé en domaines qui contiennent des droits d'accès identiques. Un domaine permet de réaliser toute une gamme de politiques de sécurité, depuis l'accès complet aux ressources jusqu'à la sandbox.

4.5.1 Contrôle d'accès

Il s'effectue au niveau d'une classe ou d'un groupe de classes (domaine de protection). Des fichiers, des ressources réseau, une fenêtre graphique constituent des ressources typiques.

Politique d'accès aux ressources

Une politique de sécurité est représentée par une classe `java.security.Policy`. Elle met en relation un certain nombre de propriétés du code qui s'exécute et des permissions qui lui sont accordées.

Modèle d'autorisation

Une politique d'accès est représentée par une liste d'entrées à deux éléments: du code identifié par ses privilèges (origine et clés publiques) et un ensemble de permissions qu'il détient. Le modèle d'accès de Java est donc à base de capacités.

Le code est caractérisé par son origine (une URL) et un ensemble de clés publiques qui doivent correspondre aux clés privées qui ont été utilisées pour le signer [5]. Ce code détient un certain nombre d'objets de type `Permission` comprenant la ressource dont la permission contrôle l'accès, et l'opération permise. On dispose d'un type de permission par type de ressource.

Pour une entrée donnée de la politique de sécurité, l'accès sera accordé au code si les informations relatives à son origine et sa signature sont cohérentes avec celles spécifiées dans la politique de sécurité.

Le modèle de sécurité de Java repose sur trois éléments: le `ClassLoader` charge le code mobile dans l'espace d'adressage approprié. Le byte-code est ensuite vérifié. Le `SecurityManager` gère enfin dynamiquement la politique de sécurité: sandbox, accès total, ou domaine de protection.

Domaine de sécurité

Java fournit deux types de domaines de sécurité (JDK 1.2):

- Le **domaine de protection** relève du niveau applicatif. Chaque classe appartient à un domaine de protection et à un seul. Les permissions sont accordées au niveau d'un domaine et non au niveau d'une classe. Pour contrôler l'accès à un objet, on examine le domaine auquel il appartient et on en déduit les permissions appropriées. La gestion des domaines de protection est implicite.

4. Dans un contexte réparti, cette classe est étendue. On obtient le `RMISecurityManager`.

5. Ces privilèges ne sont valables qu'à l'intérieur d'un thread. Les droits d'accès réels accordés à un thread correspondent à l'intersection des droits accordés par les différents domaines de protection qu'il traverse.

– Le `ClassLoader` se situe au niveau système. Il définit une unité de protection supplémentaire en définissant son propre espace de nommage. Le domaine système associé au `ClassLoader null` dispose de privilèges spéciaux pour accéder aux ressources physiques.

4.5.2 Sécurisation des communications

– Avec le JDK 1.1 est introduite la JCA (Java Cryptography Architecture) qui peut accueillir diverses implémentations pour la cryptographie à travers une API générique. Le JDK 1.1 fournit des algorithmes de signature, de condensation de messages et de génération de clés.
– Le JDK 1.2 propose en plus des services de stockage de clés, de gestion de certificats, de représentations génériques de clés, et un algorithme pour la génération de nombres aléatoires. L'outil `keytool` regroupe la majeure partie des fonctionnalités précédentes.
– Une extension au JDK, la JCE (Java Cryptographic Extension) fournit également des services de chiffrement, d'échange de clés et de codes d'authentification MAC (Message Authentication Code).

Il est enfin possible depuis le JDK 1.2 d'utiliser Java RMI par dessus un protocole de transport sécurisé du type SSL.

4.5.3 Administration d'une politique de sécurité

La gestion d'un domaine de sécurité est implicite. Par contre la politique de sécurité se veut facilement configurable: le fichier de politique de sécurité est facilement modifiable à l'aide de l'outil d'administration `PolicyTool`.

Chapitre 5

Join

5.1 Références bibliographiques

C. Fournet, G. Gonthier, The reflexive CHAM, and the join calculus, POPL'96 (1996) 372-385.

C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, D. Rémy, A calculus of mobile agents, CONCUR'96, LNCS 1119 (1996) 406-421.

J.-J. Lévy, Some results in the Join-calculus, TACS'97, LNCS 1281 (1997) 233-249.

5.2 Présentation générale

Le Join-calcul avec localités (DJoin) est un modèle pour la programmation mobile, incluant des caractéristiques explicites de colocalisation des récepteurs et de défaillance partielle, et conçu pour autoriser une implémentation distribuée. Ce modèle est doté d'une sémantique chimique, conforme à la machine abstraite réflexive RCHAM.

5.3 Critère "distribution"

5.3.1 Entités de base pour la distribution

Entités de base

Les entités de base du DJoin sont les messages, les *définitions* (jouant le rôle de règles passives), les *def-processus* (processus contenant une définition), les *patterns* de messages joints et les *solutions* (multi-ensemble de def-processus, multi-ensemble de règles actives).

Manifestations de domaines

Le concept de domaine prend l'aspect d'une localité dans le DJoin. Une localité se manifeste par une définition de localité (loc-définition), et par les concepts de *solution localisée* (sur une localité) et de *solution distribuée* (composition parallèle de solutions localisées).

Noms/références

Le DJoin manipule des noms de canaux et des noms de localité, régis par une règle de portée syntaxique. Un nom de localité désigne une localité unique, à l'intérieur de sa portée.

La portée d'un nom est distribuée, au sens où un nom peut être référencé dans une localité distante de la localité hébergeant ce nom. Cela implique un mécanisme d'infrastructure gérant la résolution globale des noms. Plusieurs processus peuvent recevoir sur un même nom, mais ils sont dans ce cas colocalisés (cf. Sémantique de la réception locale).

5.3.2 Concept de domaine

Topologie de l'espace des domaines

Les solutions localisées constituent une arborescence. L'ensemble des solutions localisées constitue une partition des récepteurs. Le concept d'éther entre les solutions localisées n'est pas explicité. Du fait de l'unicité globale des noms de localité, de chaque localité on peut en désigner une autre directement, par son nom. La structure hierarchique, en arbre, des localités est pertinente pour la migration et les défaillances, mais non pour la communication.

Sémantique des domaines

Une localité du DJoin, à l'exception de la localité racine, est une unité de migration et de défaillance. Une localité peut migrer de manière atomique, et la migration d'une localité entraîne la migration de toutes ses localités filles. Une localité peut défaillir, et entraîner ainsi la défaillance de tous les processus qu'elle abrite et de toutes les sous-localités filles. Une localité défaillante ne peut abriter aucune réaction, recevoir aucun message ni aucune localité migrante. La prise en compte des défaillances partielles est une facette cruciale de la sémantique des localités en Join-calcul.

La localité dans le Join est aussi une unité d'exécution. De plus la communication peut avoir lieu à l'intérieur d'une localité, mais la localité n'est pas l'unité de communication, au sens où celle-ci peut avoir lieu aussi entre domaines distincts.

Observabilité/contrôlabilité des domaines

Une localité peut être marquée "vivante" ou "morte", et peut être stoppée (de manière subjective, c'est à dire par une instruction contenue dans la localité elle-même). On peut observer (de manière objective, de l'extérieur) l'état d'un domaine.

De plus, comme la création de localité peut être imbriquée dans d'autres constructions du langage, celle-ci se fait de façon dynamique, ce qui donne une forme de "contrôlabilité" sur les domaines.

5.3.3 Communication locale

Désignation locale

La désignation locale d'un canal est directe (par le nom primitif).

Politique de communication locale

La communication locale est asynchrone (au sens où l'émission d'un message n'est pas bloquante), mais avec synchronisation de plusieurs messages simultanés, polyadique et univoque (à l'exécution, un seul processus reçoit effectivement sur un nom donné). Les arguments transmis sont des noms de canaux ou des noms de localités.

Sémantique de la réception

Plusieurs processus peuvent recevoir sur un nom donné (mais ils sont dans ce cas colocalisés sur une même localité). La réception locale sur un nom est statique (il n'y a pas d'appropriation de récepteurs par la communication). De plus les "join-patterns" sont répliqués. Un nom reçu devient connu, et donc utilisable comme cible d'un message.

5.3.4 Communication distante

Comme les noms définis dans un "join-pattern" sont globalement uniques, il n'y a pas de différence entre communication locale et communication distante, sauf que cette dernière peut-être empêchée par la défaillance d'un des sites concernés.

Désignation distante

La désignation d'un canal distant s'effectue de façon directe (par le nom primitif). Un nom est défini dans une localité unique, ce qui rend la désignation distante transparente à la localisation. De même, la désignation d'une localité distante s'effectue par le nom primitif de la localité.

Politique de communication distante

La communication distante est asynchrone, et s'effectue en deux étapes atomiques: la sortie du domaine courant et l'entrée dans le domaine distant (première étape), puis la consommation à proprement parler du message dans la localité cible, si celui-ci trouve les partenaires nécessaires à la formation d'un "join-pattern" (deuxième étape). Dans les deux cas la transition n'est possible que si les domaines concernés sont vivants (en fait un message peut entrer dans une localité défaillante).

Routage logique

Le routage dans l'arborescence des localités est spontané (et donc pas explicité), du fait du principe de communication distante transparente à la localisation.

5.3.5 Domaines et défaillances

Les défaillances considérées sont des défaillances permanentes, et s'appliquent uniquement aux localités (hors racine). La défaillance d'une localité implique la défaillance de toutes les sous-localités de cette localité. Une localité défaillante ne peut émettre aucun message, ni être à la source d'aucune migration vers une autre localité. Par contre, un message ou une localité peuvent migrer à destination d'une localité défaillante. La défaillance d'une localité peut être détectée depuis n'importe quelle autre localité non défaillante.

Le comportement réversible qui consiste à "activer" une définition de localité reste en principe possible même si cette localité est défaillante.

5.4 Critère "mobilité"

5.4.1 Nature des entités migrantes

Dans le Join seules les localités peuvent migrer: un domaine se déplace pour devenir un sous-domaine d'une localité désignée comme destinataire de la migration. Ceci a pour effet de modifier la topologie du réseau des domaines, mais n'a pas d'influence sur la communication, sauf à travers les défaillances: la migration d'une localité vers une autre a seulement pour effet de rendre dépendante, en ce qui concerne les défaillances, la localité source de la localité but; à l'inverse, on peut ainsi échapper à la défaillance (future) de la localité mère que l'on quitte.

Il n'y a pas de possibilité pour un processus s'exécutant dans une certaine localité, ou pour une définition qui y est localisée, de quitter cette localité, sauf si ce processus est un message destiné à un nom défini ailleurs. Ceci pour mettre en œuvre la caractéristique essentielle du Join, qui est qu'une définition est associée à une localité unique et statiquement déterminée.

5.4.2 Expression de la migration

Elle prend la forme d'une instruction $\mathsf{go}(a,\kappa)$ où a est la localité de destination et κ une continuation. L'effet de cette instruction est de déplacer la localité englobante (celle qui contient cette instruction) pour en faire une sous-localité de a, en déclenchant à l'arrivée le signal κ. C'est donc une forme subjective de migration.

Cette instruction est asynchrone par rapport à l'entité dont elle provoque la migration. En particulier, si des messages sont destinés à la localité concernée par un $\mathsf{go}(a,\kappa)$, le moment où ces messages sont traités (et le traitement lui-même) est a priori indépendant du moment où a lieu la migration effective. Si le contenu de la localité est un programme séquentiel, on peut synchroniser son exécution avec la migration, en utilisant la continuation κ.

5.4.3 Mobilité en cours d'exécution

L'instruction de migration peut être utilisée comme n'importe quelle autre construction du langage, et donc la migration d'une localité peut avoir lieu au cours de l'exécution des calculs dans cette localité. On peut dans une certaine mesure contrôler le comportement de l'entité migrante (en bloquant un processus à effectuer après déplacement), mais non le flux des messages entrants.

5.4.4 Migration d'un état associé

Il n'y a pas véritablement de notion d'état associé à une localité. On peut dire que le contexte de l'entité migrante – si l'on entend par là les définitions sur lesquelles la localité possède une référence, mais qui ne sont pas dans la localité elle-même – reste fixe.

5.4.5 Entretien de références distantes

Une localité qui migre conserve les liens qu'elle entretient avec son contexte, puisque ceux-ci se matérialisent simplement par des noms, qui suffisent à établir une communication distante. Cette préservation des liens se fait de manière complètement transparente.

5.5 Critère "sécurité"

Il existe une version du JOIN qui intègre sous une forme abstraite des primitives de chiffrement et déchiffrement de message, analogue au spi-calcul (voir la fiche sur Pict). Nous n'avons pas étudié cette version.

Chapitre 6

Kali Scheme

6.1 Références bibliographiques

H. Cejtin, S. Jagannathan, R. Kelsey, Higher-order distributed objects, ACM ToPLaS Vol. 17, No. 5 (1995) 704-739.

6.2 Présentation générale

Kali Scheme est une extension du langage Scheme qui permet, dans un cadre réparti, la transmission de toute entité du langage, y compris les valeurs d'ordre supérieur comme les clôtures et continuations. Ainsi, Kali Scheme est un langage pour la mobilité "forte".

6.3 Critère "distribution"

6.3.1 Entités de base pour la distribution

Entités de base

Toutes les entités manipulées dans le langage Scheme sont susceptibles d'être transmises par message. Les entités spécifique que Kali Scheme implémente sont les espaces d'adressage et les délégués ("proxies"). Les proxies ont un nom dont la portée peut s'étendre sur plusieurs espaces d'adressage, mais ont une valeur propre à chaque espace où ils sont définis.

Manifestations de domaines

Un "domaine" dans Kali Scheme est un espace d'adressage. C'est une notion "de première classe", qu'on peut utiliser au même titre que les autres constructions du langage.

Noms/références

Contrairement à Lisp, Scheme implémente la portée lexicale. Les espaces d'adressage sont supposés avoir un nom globalement unique.

6.3.2 Concept de domaine

Topologie de l'espace des domaines

A la création, un espace d'adressage ouvre un canal de communication avec tous les autres "domaines" existants. On peut donc considérer que l'espace des "domaines" en Kali Scheme est plat, sans imbrication: d'un domaine on a accès à tous les autres, sans passer par une hiérarchie.

Sémantique des domaines

Un espace d'adressage est une unité d'exécution, dans laquelle des "threads" s'exécutent en parallèle (i.e. en entrelacement). C'est aussi une unité de communication – voir ci-dessous. Par contre il n'y a pas de considération de défaillance ou de sécurité. Les deux opérations primitives sont la création d'un espace d'adressage, avec une adresse Internet, et la consultation du nom de l'espace dans lequel on se trouve.

6.3.3 Communication locale

Désignation locale

Directe: un identificateur est utilisé comme il peut l'être dans n'importe quel langage de programmation standard.

Politique de communication locale

La communication à l'intérieur d'un espace d'adressage se fait par variables partagées. Plus généralement, un "domaine" dans Kali Scheme étant un système de "threads", toute primitive représentant une forme de communication entre "threads" (par exemple utilisation de verrous) fournit une notion de communication locale.

Sémantique de la réception

C'est l'affectation d'une valeur à une variable.

6.3.4 Communication distante

Il n'y a pas de communication distante.

6.4 Critère "mobilité"

6.4.1 Nature des entités migrantes

On peut a priori envoyer n'importe quelle entité du langage d'un espace d'adressage à un autre, à travers le canal établi entre ces deux "domaines". Par exemple, un espace d'adressage étant une entité "de première classe", il n'est pas exclu a priori que l'on puisse l'envoyer dans un message, c'est à dire le faire migrer vers un autre "domaine". Toutefois la sémantique d'une telle opération n'est pas décrite – a priori il ne devrait rien se passer de particulier, puisque les espaces d'adressage forment un graphe fortement connexe.

6.4.2 Expression de la migration

Elle prend la forme d'une instruction d'envoi de message, à travers le canal reliant deux espaces d'adressage, dont la syntaxe est la suivante:

(`send-message type message channel lock`)

Il y a plusieurs types de messages possibles, le plus important étant le type `run`, qui provoque l'exécution – ou plutôt l'évaluation – du contenu `message` dans la localité désignée comme l'extrémité du canal `channel`. Le verrou sert à empêcher l'envoi "simultané" de messages depuis le même espace d'adressage (la composition du message n'est pas une opération atomique).

La topologie du "réseau" étant celle d'un graphe fortement connexe, il n'y a pas de routage à gérer au niveau des programmes. La migration, c'est à dire l'envoi de message, est asynchrone. Toutefois on peut naturellement utiliser une technique de continuations pour programmer une version synchrone, bloquant la suite de l'exécution du programme.

Cette "instruction" n'est en fait pas une primitive: elle repose sur des opérations `encode`, qui compose ce qui va être effectivement transmis à partir du `type` et du `message`, et `channel-write` qui envoie effectivement le message. La sémantique de ces opérations n'est pas précisée. Normalement le contenu du message est retransmis par copie, mais certaines entités, et notamment les "proxies", sont transmises par référence – on en transmet le nom mais pas la valeur.

6.4.3 Mobilité en cours d'exécution

Comme on a accès au niveau du langage à une primitive de migration qui peut s'appliquer à n'importe quelle entité, on peut réaliser la mobilité au cours des calculs, par imbrication des telles instructions. L'envoi de message est bloquant par rapport à son contenu, on a donc un certain contrôle sur ce que l'on veut faire migrer. Toutefois, il faut tempérer cette affirmation, car la détermination de ce qui migre exactement ne semble pas être accessible au niveau du langage (voir ci-dessous).

6.4.4 Migration d'un état associé

L'exécution d'une instruction de migration provoque en fait le déplacement d'une structure d'exécution, c'est à dire d'une partie du contexte dans lequel le contenu `message` s'évalue. En particulier, on peut provoquer la migration de continuations, qui contiennent non seulement un "thread" mais aussi un état associé.

Il semble toutefois que la sémantique de la migration dépende du type de l'entité migrante. En particulier, les continuations sont communiquées de manière paresseuse – des proxies étant créés pour les portions non-transmises: "*sending a thread's continuation involves sending only its top few frames, together with a handle stored at the base of the sent continuation. The value slot of this handle contains a reference to the rest of the continuation stack*". On doit aussi définir un traitement pour l'exception qui est levée lorsqu'on accède à une référence vers une portion non transmise.

La sémantique de la migration n'est donc définie que par l'implémentation (*cf.* the *top few frames...*), et si on peut parler de migration forte, il est difficile de déterminer jusqu'à quel point elle l'est: on ne sait pas très bien quelle portion du contexte d'exécution d'un "thread" migre avec le contrôle et le code. On ne sait pas non plus ce qui se passe par exemple pour les variables que ce "thread" pourrait partager avec d'autres.

6.4.5 Entretien de références distantes

Le langage offre, outre les primitives associées aux espaces d'adressage, une notion de "proxy" (un peu non standard), qui est un nom connu de plusieurs "domaines", mais avec une valeur distincte dans chacun d'eux (cette valeur ne consiste pas forcément en la retransmission de requête vers un objet principal). Avec cette notion, et le fait qu'on peut connaitre le site créateur d'un "proxy", on peut établir et maintenir des références distantes.

On peut dire qu'une partie du contexte de l'entité migrante est fixe, avec établissement de "proxies" (en particulier pour les éléments du contexte qui sont eux-mêmes des proxies), une autre étant mobile par réplication, sans toutefois pouvoir être plus précis à ce sujet.

6.5 Critère "sécurité"

L'aspect sécurité n'est pas abordé. Les auteurs mettent toutefois en avant l'idée que, au moins par rapport à d'autres langages, le contrôle de types (à l'exécution) et la portée lexicale des noms offrent certaines garanties de ce point de vue. L'idée de sécuriser les canaux de communication entre les domaines par des fonctions cryptographiques est évoquée, mais vue comme orthogonale à la migration en elle-même.

Chapitre 7

Obliq

7.1 Références bibliographiques

L. Cardelli, A language with distributed scope, Computing Systems Vol. 8, No. 1 (1995) 27-59.

7.2 Présentation générale

Obliq est un langage à objets fondé sur l'idée de "transparence du réseau": la sémantique d'un programme, qui peut être constitué d'un système d'objets répartis sur des sites distants, ne dépend pas de la localisation physique de ses composants. Cela suppose que la transmission d'une entité d'un site à l'autre n'affecte pas les références qu'elle peut entretenir; en particulier, la règle de portée lexicale doit être valide au niveau d'un système réparti.

7.3 Critère "distribution"

7.3.1 Entités de base pour la distribution

Entités de base

Les entités de base de l'implémentation d'Obliq sont les sites, les adresses mémoire – qu'on appellera ici "variables", car on peut mettre à jour leur valeur –, les valeurs et les "threads". Parmi les valeurs on trouve les objets et les clôtures. Une clôture est une paire formée d'un code et d'une table des valeurs des identificateurs libres. Un objet est une collection de champs dont la valeur peut être une méthode, un alias ou n'importe quelle autre valeur.

Manifestations de domaines

Les sites sont complètement implicites, et ne sont pas accessibles au niveau du langage.

Noms/références

Les noms sont les identificateurs, au sens usuel. Chaque nom a une portée locale; sa valeur peut être une référence qui pointe vers une adresse dans un site unique où est localisée la valeur effective du nom. Les références (adresses mémoire ou références distantes) ne sont pas manipulables directement au niveau du langage.

7.3.2 Concept de domaine

Topologie de l'espace des domaines

Elle n'est pas explicitée, mais semble être "plate": rien n'indique qu'un site puisse être contenu dans un autre.

Sémantique des domaines

Un site dans Obliq est une unité d'exécution: c'est un espace d'adressage dans lequel s'exécutent des "threads" concurrents.

7.3.3 Communication locale

Il n'y a pas à proprement parler de notion de communication dans Obliq. On peut considérer que la communication consiste à exécuter des opérations usuelles comme l'affectation de valeur à une variable, l'appel à une procédure ou une fonction, l'invocation d'une méthode d'un objet, etc. Cette dernière forme est souvent conçue comme un "envoi de message" dans les langages à objets.

Désignation locale

Directe, par identificateur.

7.3.4 Communication distante

Désignation distante

On peut établir une référence sur une entité distante via un serveur de noms commun, par enregistrement de cette entité sous un certain nom et consultation de ce nom. Tout lien sur la référence ainsi acquise est ensuite utilisé comme n'importe quel autre nom.

Politique de communication distante

Toute valeur (ainsi que toute référence distante), sauf les objets, peut être transmise à travers le réseau, par copie jusqu'au point où elle contient des adresses mémoire, qui sont transformées en références distantes.

Lorsqu'un champ (dont la valeur n'est pas une méthode) d'un objet est sélectionné à distance, sa valeur est transmise vers le site d'où émane la sélection; de cette façon on peut effectuer du "code fetching", si la valeur du champ est une procédure par exemple. A l'inverse, l'invocation de méthode résulte en son évaluation dans le site de l'objet invoqué; c'est une invocation synchrone, l'appelant étant bloqué dans l'attente d'un résultat.

Lorsque la valeur d'un champ (qui peut être une méthode) est mise à jour à distance, la nouvelle valeur (qui est une clôture s'il s'agit de modifier une méthode) est transmise et installée dans l'objet (on peut ainsi réaliser une forme de "code shipping"). Le clonage d'un objet (possiblement distant) résulte en la création d'une copie locale de l'objet; la valeur de ses différents champs est transmise selon la discipline mentionnée ci-dessus.

Enfin on peut transformer un champ d'un objet en un alias d'un autre – toutes les opérations sur ce champ (y compris la mise à jour) sauf la création d'un nouvel alias sont redirigées vers l'autre objet. On peut ainsi rediriger entièrement les opérations affectant un objet vers un autre, transformant ainsi le premier en "délégué" du second.

7.4 Critère "mobilité"

7.4.1 Nature des entités migrantes

Ce sont les valeurs, excepté les objets, c'est à dire tout ce qui est transmissible à travers le réseau. En particulier, on peut faire migrer une procédure, ou plus exactement sa clôture. La migration se fait par copie (partielle), sauf dans le cas où elle est provoquée par une mise à jour.

7.4.2 Expression de la migration

Elle est indirecte, et résulte de la sémantique de la communication: la sélection d'un champ à distance par exemple provoque la migration de sa valeur, si celle-ci n'est pas une méthode, vers le site de l'appel. A l'inverse, la mise à jour d'un champ, ou plus généralement d'une variable, provoque la migration de la nouvelle valeur vers le site où réside ce champ.

La migration d'un objet s'effectue en en créant un clone dans la localité de destination, et en redirigeant les opérations qui lui sont destinées vers son clone (il faut toutefois être capable de faire cela de façon atomique).

On peut dire que la migration est objective, et bloque l'évaluation de l'entité transmise. Ainsi on a (en principe) un contrôle exact sur l'entité transmise.

7.4.3 Mobilité en cours d'exécution

Elle est possible, puisque la migration est provoquée par les instructions de base du langage (sélection/invocation, mise à jour, clonage). Toutefois, les valeurs pouvant seules migrer, la mise en œuvre du "paradigme agent" ne parait pas très aisée.

7.4.4 Migration d'un état associé

Les adresses mémoire sont attachées à un site et ne peuvent migrer. La migration d'une entité entraine celle de sa clôture, comprenant la valeur de ses identificateurs libres (qui peut être une référence distante). Il n'y a donc pas de migration d'état, du moins pas de l'état constitué au cours de l'exécution d'une entité migrante.

7.4.5 Entretien de références distantes

La caractéristique d'Obliq est de ne jamais rompre les liens entretenus par une entité migrante. Ces liens sont transformés en références distantes lorsqu'ils pointent sur une entité qui ne peut être déplacée (typiquement une adresse mémoire ou un objet).

7.5 Critère "sécurité"

Obliq ne dispose pas de véritable modèle de sécurité, mais offre deux formes de protection: la notion d'objet protégé, et celle de portée lexicale qui permet de contrôler l'accès aux ressources.

De plus, Obliq s'appuie sur les mécanismes de sécurité présents dans la bibliothèque Network Objects dont il existe une variante sécurisée appelée **Secure Network Objects** (cf. délivrable MARVEL "Document d'analyse de machines virtuelles"). Cette bibliothèque permet la réalisation d'une implémentation sécurisée d'Obliq.

7.5.1 Contrôle d'accès

Obliq admet l'objet comme unité de protection. Les ressources auquel on cherche à accéder sont les valeurs.

Modèle d'autorisation

Le contrôle d'accès aux ressources locales s'effectue à l'aide de capacités.

Obliq dispose de la notion d'**objet protégé** pour empécher un client de modifier une méthode d'un serveur ou de le cloner. Un objet protégé rejette des mises à jour, clonage, ou aliasing venant de l'extérieur mais les accepte si elles proviennent de l'intérieur de l'objet.

Pour contrôler l'accès aux ressources locales, Obliq utilise la **portée lexicale**: une procédure évaluée par un moteur d'exécution (execution engine ou représentation au niveau langage d'une invocation à distance) ne peut accéder à des objets qu'à l'intérieur de sa portée. Plus généralement, l'accès d'une procédure "nomade" à des ressources locales se fait à travers des capacités qui sont obtenues avec la coopération explicite du serveur.

Dans l'implémentation sécurisée d'Obliq mentionnée précédemment, des listes de contrôle d'accès sont utilisées en plus de capacités pour contrôler l'accès aux ressources: un objet Obliq est exporté avec la liste des entités pouvant l'importer. Cet objet est encapsulé dans un moniteur de références contenant deux méthodes: une pour accéder à l'objet, l'autre pour retourner l'identité du moniteur de références, ce qui permet au client de s'assurer de la véritable identité du serveur.

Domaine de sécurité

La portée lexicale répartie offre en Obliq une certaine abstraction d'un domaine de protection: dans le cas d'une procédure évaluée par un moteur d'exécution, le domaine de protection est consitué par la portée de la procédure.

Chapitre 8

Pict et Nomadic Pict

8.1 Références bibliographiques

D. Turner, The Polymorphic Pi-calculus: Theory and Implementation, PhD Thesis, University of Edinburgh (1995).

B. Pierce, Programming in the Pi-Calculus: A Tutorial Introduction to Pict (Pict Version 4.0), Technical Report, Indiana University, March 1997.

P. Sewell, P. Wojciechowski, B. Pierce, Location-independent communication for mobile agents: a two-level architecture, Tech. Rep. 462, Computer Lab, University of Cambridge (1998).

Page de référence de Pict:

http://www.cis.upenn.edu/ bcpierce/papers/pict/Html/Pict.html

Pour le spi-calcul, l'article de référence est:

M. Abadi, A. Gordon, A calculus of cryptographic protocols: the spi-calculus, Information and Computation Vol. 148, No. 1 (1999)1-70.

8.2 Présentation générale

Dans cette fiche on présente à la fois Pict et Nomadic Pict. Le premier est un langage basé sur le π-calcul, n'incluant pas de primitive de migration, tandis que le second propose une infrastructure et quelques primitives pour la programmation d'agents migrants, le modèle de base étant le π-calcul. Bien que l'intégration de ces deux modèles n'aie pas été réalisée à ce jour, il est clair qu'ils sont complémentaires.

Le π-calcul est un calcul de processus à base de canaux adapté à la description de systèmes concurrents dont la topologie évolue dynamiquement. C'est un calcul de processus mobiles car il permet de passer un canal de communication comme valeur sur un autre canal. Un canal sert de référence à un processus, ce qui rend ces derniers mobiles.

Pict est un langage fonctionnel de haut-niveau pour la programmation concurrente construit uniquement en terme de primitives du π-calcul. Le langage Pict est construit au-dessus d'un calcul primitif appelé **Core Pict**. Il s'agit d'un π-calcul asynchrone sans opérateur de sommation, statiquement typé, avec des tuples et des records. Le langage est constitué d'un certain nombre de règles de réécriture (sucre syntaxique) fournissant des abstractions de plus haut niveau que le calcul primitif. Il permet de définir des expressions conditionnelles, des valeurs complexes, des processus serveurs (Pict les transformera en récepteurs répliqués), etc. Le regroupement de plusieurs processus serveurs à l'intérieur d'un record permet de simuler un objet avec ses différentes méthodes. Comme on l'a dit, Nomadic Pict est propose une modélisation des agents migrants qui est destinée à compléter le langage Pict. Le modèle de Nomadic Pict peut être considéré comme une formalisation directe de (certains aspects de) la notion d'agent introduite par Telescript.

8.3 Critère "distribution"

8.3.1 Entités de base pour la distribution

Entités de base

Les entités de base sont des processus qui sont référencés par des noms de canaux. Ils peuvent être des processus lecteurs $?[x].P$, lecteurs répliqués $? * [x].P$ ou écrivains $![x].P$. Nomadic Pict complète cela par les notions de site et d'agent, qui est un processus nommé et localisé (dans un site).

Manifestations de domaines

Les domaines ne sont pas présents dans Pict car le π-calcul sous-jacent ne dispose pas de notion de localité. La notion de portée peut cependant être considérée comme une abstraction d'un domaine de sécurité.

Dans Nomadic Pict on peut considérer que les sites, qui se manifestent essentiellement par leur nom, constituent les domaines. Il n'y a pas de contruction dans le langage pour les sites: ceux-ci peuvent seulement être référencés comme "cible" pour la migration. On peut également considérer un agent comme un "domaine", contenu dans un site, et le langage offre une primitive pour introduire des agents.

Noms/références

Les canaux sont nommés et référencent des processus. L'ensemble des processus qui communiquent est non-structuré[1] L'espace de nommage de Pict est global: chaque nom de canal est supposé par définition unique. Toutefois il faut se rappeler que les programmes Pict opèrent sur un site unique, donc il n'y a pas pour Pict de différence entre "local" et "global". Dans Nomadic Pict également l'espace de nommage est global. La portée d'une référence correspond à sa portée lexicale.

Une référence est représentée par un nom de canal qui peut être créé à l'aide de l'**opérateur de restriction** ν. Le calcul présuppose un service de nommage pour créer un nom de canal unique globalement n'ayant pas été déjà utilisé. (fresh name).

8.3.2 Concept de domaine

Topologie de l'espace des domaines

Dans Pict il n'y a pas de notion de domaine, tandis que Nomadic Pict se caractérise par une structure hiérarchique à deux niveaux: les sites contiennent les agents, qui eux-mêmes contiennent les processus s'exécutant.

Sémantique des domaines

Un site de Nomadic Pict est une unité de communication: les agents présents dans un même site peuvent communiquer entre eux. À l'intérieur d'un agent, les processus s'exécutent et peuvent communiquer entre eux, par envoi de messages comme dans le π-calcul de base.

8.3.3 Communication locale

Désignation locale

Elle se fait directement par un nom de canal.

Politique de communication locale

C'est celle du π-calcul polyadique et asynchrone. Toutefois, le langage Pict est typé, et l'on peut distinguer par typage divers usages des noms de canaux: en réception seulement, en émission seulement, ou bien avec les deux capacités. À cela Nomadic Pict ajoute la possibilité d'envoyer à un agent, spécifié par son nom, un message (au sens du π-calcul) destiné à un récepteur contenu dans cet agent. Si cet envoi échoue, l'agent

1. La notion de **record** permet de grouper un ensemble de processus sous un même nom. On se rapproche ici de la notion d'objet, chaque processus jouant le rôle d'une méthode. Ces processus peuvent être rendus disponibles à l'extérieur à l'aide de ports constitués des champs du record, ce qui ressemble à la notion d'**interface** utilisée dans le monde orienté-objets. Cette interface peut être communiquée d'un processus à l'autre en utilisant le caractère polyadique du π-calcul qui permet de transmettre des records. On se rapproche alors du passage d'une référence d'un espace d'adressage à un autre que l'on trouve classiquement dans les systèmes répartis à objets.

partenaire n'étant pas localisé sur le même site, un processus qu'on peut considérer comme une "exception" est lancé.

Sémantique de la réception

Dans Pict, la réception est multiple (plusieurs processus peuvent recevoir sur un nom). Le fait que de multiples émetteurs et récepteurs tentent d'utiliser le même canal simultanément contribue à rendre le π-calcul non-déterministe. La machine abstraite Pict transforme cet indéterminisme en exécution déterministe en garantissant l'**équité** des différentes exécutions possibles, et leur **terminaison**. Lors d'une invocation, un appel sera reçu au maximum une fois par le processus récepteur. Dans Nomadic Pict, la communication suit le même schéma, mais est confinée à l'intérieur des agents.

En π-calcul, l'**opérateur de choix** peut être utilisé comme barrière de synchronisation. Dans le langage Pict, l'opérateur de choix a été abandonné pour simplifier à la fois la sémantique formelle du calcul et l'implémentation [2].

8.3.4 Communication distante

Cette notion n'a pas de sens en Pict, puisque le modèle sous-jacent n'est pas réparti: il n'y a pas de différence entre communication locale et communication distante car on ne dispose pas de notion de localité.

Dans Nomadic Pict, il y a au niveau du langage une construction qui permet d'envoyer un message à un agent de façon "globale", c'est à dire sans connaitre la localité du destinataire, mais l'un des objectifs de Nomadic Pict est de montrer que cette construction n'est pas primitive, et s'implémente avec les primitives de communication locale offertes par le langage. On peut donc dire que Nomadic Pict ne repose pas sur un mécanisme de communication distante.

Routage logique

Le routage est implémenté par Nomadic Pict, au sens où un message envoyé à un agent non localisé est géré – i.e. envoyé dans le site où se trouve l'agent – par la couche de bas niveau, reposant sur la communication purement locale, du langage. Cette implémentation n'est pas répartie: on utilise un serveur de requêtes unique pour garder la trace des agents migrants.

8.4 Critère "mobilité"

8.4.1 Nature des entités migrantes

Dans Pict il n'y a pas de migration. Dans Nomadic Pict, l'entité migrante est l'agent, qui peut être vu comme un domaine nommé contenant des processus Pict s'exécutant. Il y a également une autre forme de migration, qui consiste à envoyer un message (au sens du π-calcul) d'un agent vers un autre si le destinataire se trouve sur le même site que l'agent émetteur. Donc on peut dire que les entités migrantes entre sites sont les agents, et que celles qui migrent entre agents sont les messages.

8.4.2 Expression de la migration

La migration s'exprime sous la forme d'une instruction migrate to $s \rightarrow P$, qui a pour effet d'envoyer l'agent contenant cette instruction vers le site s; le processus P sera alors exécuté comme part de l'agent migrant. Il s'agit donc de migration "subjective", asynchrone par rapport à l'entité migrante.

2. Pierce *et al.* décrivent un opérateur dit de **choix répliqué** qui peut être utilisé pour contrôler l'exécution concurrente de méthodes à l'intérieur d'un objet. Un processus du type:

$$*P = \text{read?}[r].(\cdots \mid P) \; + \; \text{update?}[n,r].(\cdots \mid P)$$

peut être implémenté à l'aide de cet opérateur de choix répliqué qui assurera l'équité des exécutions entre `read` et `update` d'une itération à l'autre ce qui n'est pas possible avec un opérateur de choix simple si le rythme des invocations est différent pour `read` et `update`.

8.4.3 Mobilité en cours d'exécution

L'instruction de migration peut être utilisée comme n'importe quelle autre construction du langage, et donc la migration d'un agent peut avoir lieu au cours de l'exécution des calculs de cet agent. Pour synchroniser le comportement de l'agent avec la migration, il faut utiliser la continuation de l'instruction de migration pour contrôler les autres processus présents dans l'agent – mais on ne peut contrôler l'arrivée de nouveaux messages dans l'agent migrant.

8.4.4 Migration d'un état associé

Il n'y a pas véritablement de notion d'état associé à un agent. Le "contexte" d'un agent – si l'on entend par là les récepteurs sur lesquels l'agent possède une référence – reste fixe (sauf la partie contenue dans l'agent, bien entendu).

8.4.5 Entretien de références distantes

L'agent migrant conserve les liens qu'il possède, puisque ceux-ci sont matérialisés par un nom de canal. Toutefois, ces liens distants ne peuvent être utilisés directement pour la communication. Si un agent a migré alors qu'il voulait communiquer avec un agent voisin, une sorte d'exception est lancée: un message ne trouvant pas localement de destinataire n'est donc pas nécessairement perdu. En particulier ceci permet de programmer une couche de communication globale (via un serveur central).

8.5 Critère "sécurité"

Pict ne dispose pas de mécanismes de sécurité. Il est cependant intéressant de considérer une extension sécurisée du π-calcul ou **spi-calcul** dévéloppé par Abadi et Gordon qui pourrait être utilisée pour définir une extension sécurisée de Pict ou SPict.

Le π-calcul permet de créer des canaux de communication et de les passer d'un processus à l'autre e.g. d'un serveur d'authentification vers un client. Les règles de portée garantissent que l'environnement d'un protocole (e.g. une entité malveillante) ne pourra avoir accès à un canal que si cet accès lui est fourni explicitement. La portée est donc un ingrédient fondamental pour les protocoles de sécurité. Les primitives de chiffrement ne sont pas faciles à représenter en π-calcul, ce qui nécessite une extension intégrant de telles primitives. On obtient alors le spi-calcul, qui permet d'analyser des protocoles de sécurité en termes d'intégrité et de confidentialité. Les caractéristiques principales du spi-calcul sont:

- L'utilisation du mécanisme de portée du π-calcul.
- La définition de l'environnement d'un protocole de sécurité comme un processus arbitraire du calcul.
- La représentation des propriétés de sécurité (confidentialité, intégrité) à l'aide de relations d'équivalences. Leur signification informelle est la suivante: deux protocoles mis en relation ne sont pas distinguables, même quand une entité malveillante se trouve dans leur environnement.

8.5.1 Identification et authentification

Le spi-calcul dispose de l'expressivité nécessaire pour pouvoir décrire des protocoles d'authentification. Divers protocoles d'authentification (mutuelle et à travers un tiers) sont analysés en spi-calcul par Abadi et Gordon dans l'article de référence.

8.5.2 Contrôle d'accès

Le contrôle d'accès en spi-calcul se fait au niveau des canaux de communication.

Modèle d'autorisation

Un privilège en spi-calcul est constitué d'un nom de canal. C'est une représentation abstraite d'un secret quelconque (e.g. une clé) que les primitives du π-calcul vont permettre de détenir ou de communiquer. Le nom de canal joue le rôle d'une capacité. Deux caractéristiques du π-calcul sont intéressantes pour la sécurité:

- **La restriction:** les possibilités de communication d'un processus sont déterminées par les canaux qu'il connaît. L'opérateur de restriction permet à seulement certains processus d'avoir accès à un canal. Il peut être utilisé pour réaliser du contrôle d'accès.
- **L'extension de portée:** les processus du calcul sont mobiles au sens où leurs possibilités de communication peuvent changer au cours du temps: ils peuvent apprendre de nouveaux noms de canaux par extension de portée, ce qui peut être utilisé pour communiquer des secrets, ou pour étendre des droits d'accès. Le concept de portée est également une forme d'abstraction d'un domaine de sécurité.

Domaine de sécurité

Le concept de portée joue le rôle en spi-calcul de domaine de sécurité. Les noms de canaux contenus à l'intérieur d'une portée ne peuvent pas être divulgués à l'extérieur de cette portée sauf explicitement par transmission de ces noms (et extension de portée). Cette notion de domaine garantit à la fois la confidentialité et l'intégrité de ces noms: le spi-calcul modélise les attaques possibles comme venant de l'extérieur du système i.e. le système global est vu comme le décrit l'équation:

$$\text{Système global} = \text{Cryptanalyste} \mid \text{Système}$$

où le cryptanalyste est un exemple d'utilisateur malveillant. Ces garanties doivent être nuancées car le spi-calcul ne prend pas en compte des attaques venant de l'intérieur du système. L'extension de portée peut également modéliser l'entrée d'un processus dans un domaine de sécurité.

8.5.3 Sécurisation des communications

Le spi-calcul permet de garantir des propriétés de confidentialité et d'intégrité pour des cryptosystèmes à clé secrète ou publique. Ce calcul enrichit le π-calcul à l'aide de deux termes, un pour le chiffrement, l'autre pour le déchiffrement. L'opérateur de restriction peut être utilisé pour créer de nouvelles clés qui ne peuvent pas être devinées.

Écrire un protocole de sécurité en spi-calcul semble plus difficile que de l'exprimer par une description informelle traditionnellement donnée dans la littérature. Cela reste équivalent à l'écrire dans n'importe quel langage de programmation doté des bibliothèques de communication et de chiffrement adéquates, mais en bénéficiant en plus d'une sémantique précise offerte par le calcul, et d'une modélisation assez générale des attaques du système de sécurité relativement indépendante de ce système.

Chapitre 9

$\pi_{1\ell}$-calcul

9.1 Références bibliographiques

R.Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In Proc. Coordination 97, Springer Lect. Notes in Comp. Sci. 1282, 1997. Extended version appeared as RR-INRIA 3109.

R. Amadio. On Modelling Mobility. A paraitre dans TCS.

9.2 Présentation générale

Le $\pi_{1\ell}$-calcul est une version asynchrone typée du π-calcul, formalisant le concept de *localités*, et fournissant une sémantique abstraite pour la communication distante, la migration et la défaillance. Une caractéristique de ce calcul est qu'il assure l'unicité globale des récepteurs sur les canaux, la communication étant indépendante de la localisation.

9.3 Critère "distribution"

9.3.1 Entités de base pour la distribution

Entités de base

Les entités de base du $\pi_{1\ell}$-calcul sont les processus, les localités et les configurations. Un processus est un composé (en particulier par la mise en parallèle) de capacités à recevoir des messages et d'envoi de messages. L'opérateur de restriction permet de délimiter la portée d'un nom. Une localité contient des processus s'exécutant en parallèle. Les configurations, représentant l'état d'un réseau, sont constituées de localités et de messages en parallèle, un message pouvant être une émission au sens du π-calcul, un processus migrant, ou une instruction de contrôle de l'état d'une localité.

Manifestations de domaine

Un domaine du $\pi_{1\ell}$-calcul porte le nom de *localité*. Une localité du $\pi_{1\ell}$-calcul se manifeste par un processus particulier (appelé dans la suite *loc-processus*), et par des *processus localisés* i.e. abrités par une localité. Un loc-processus peut recevoir des messages particuliers, dédiés au contrôle de la localité, sur un port unique portant le nom de la localité (voir ci-dessous).

Noms/références

Le $\pi_{1\ell}$-calcul manipule deux sortes de noms: les noms de canaux et les noms de localités. Les uns et les autres sont régis par une règle de portée syntaxique. Un nom de localité désigne, dans sa portée, une entité unique. De même un nom de canal désigne un récepteur unique. La portée d'un nom est distribuée, au sens où un nom peut être référencé dans une localité distante de la localité hébergeant ce nom. Cela implique un mécanisme d'infrastructure gérant la résolution globale des noms.

9.3.2 Concept de domaine

Topologie de l'espace des domaines

L'espace des localités est non hiérarchique, et constitue l'espace feuille de l'arbre des configurations. L'ensemble des localités constitue une partition de l'ensemble des récepteurs. Entre les localités, il existe un éther dans lequel circulent des messages. Chaque localité est supposée directement accessible depuis les autres.

Sémantique des domaines

Une localité du $\pi_{1\ell}$-calcul est une unité de défaillance, au sens où la défaillance d'une localité conditionne la défaillance de tous les processus qu'elle abrite. C'est aussi une unité d'exécution. La communication peut avoir lieu à l'intérieur d'une localité, mais la localité n'est pas l'unité de communication, au sens où celle-ci peut avoir lieu aussi entre domaines distincts.

Observabilité/contrôlabilité des domaines

Une localité possède un état (Run/Stop) et peut être observée et contrôlée à distance via le loc-processus correspondant. L'observation est un test de l'état, alors que le contrôle prend la forme d'un arrêt. On peut stopper une localité ou observer son état aussi bien depuis une localité distante que depuis la localité elle-même, pourvu qu'elle soit vivante.

9.3.3 Communication locale

Désignation locale

La désignation locale d'un canal du $\pi_{1\ell}$-calcul est directe, i.e. sollicite le nom primitif du canal.

Politique de communication locale

La communication locale est asynchrone, polyadique et univoque (à l'exécution, un seul processus reçoit effectivement sur un nom donné). Les arguments transmis sont des noms de canaux ou des noms de localités.

Sémantique de la réception

La réception locale sur un nom est persistante (un récepteur reste récepteur potentiel après la communication), et statique (il n'y a pas d'appropriation de récepteurs par la communication). Enfin, un nom reçu devient connu, et donc utilisable comme cible d'un message.

9.3.4 Communication distante

Désignation distante

La désignation distante d'un canal est directe (nom primitif). Ce principe de désignation transparente à la localisation permet néanmoins de déterminer sans ambiguité la localisation du récepteur d'un message, puisque ce récepteur est globalement unique. Son nom est donc une adresse absolue dans la configuration[1]. Il en va de même pour la désignation de localité.

Politique de communication distante

La communication distante est asynchrone. Elle s'effectue en deux étapes atomiques: le message sort du domaine courant, à condition que ce domaine soit "vivant" (étape 1), puis est acheminé vers le domaine, a priori distant, contenant le récepteur et y est consommé (étape 2). Rien ne distingue en fait la communication distante de la communication locale, car ces deux étapes valent même si la localité destinataire coïncide avec la localité de départ (si celle-ci est vivante).

1. ceci n'est vrai toutefois que dans la version du calcul présentée dans le premier article, où un processus contenant un récepteur sur un canal public n'est pas autorisé à migrer, voir ci-dessous.

Routage logique

Du fait du principe de communication transparente à la localisation, le routage est spontané (et donc non explicité).

9.3.5 Domaines et défaillances

Le calcul formalise la détection de défaillances par arrêt et les défaillances transitoires de localités, via le loc-processus associé. Une localité arrêtée bloque tous les processus qu'elle abrite, i.e. bloque l'émission des messages à partir de cette localité.

La défaillance d'une localité peut être testée depuis une localité distante (message *ping*). Le *ping* est assimilé à l'interrogation d'un oracle. Le premier type de détecteur formalisé dans le $\pi_{1\ell}$-calcul est le détecteur de défaillance parfait. Dans ce cas, l'oracle est supposé sans failles. Cela sous-entend que la réponse à un *ping* est correcte, y compris dans le cas où le processus de localité correspondant est défaillant. Le calcul permet aussi de formaliser l'exécution d'un détecteur non parfait, en introduisant deux états supplémentaires dans les loc-processus.

Une localité en marche peut être arrêtée depuis une localité distante, par l'émission d'un message *stop*. Le calcul prévoit un second message d'arrêt: *maystop*, se comportant comme *stop* ou comme 0 de manière indéterministe, et permettant de décrire un processus "susceptible de panne".

9.3.6 Domaines et sécurité

La sécurité dans le $\pi_{1\ell}$-calcul se limite aux droits à communiquer, et est capturée par la portée syntaxique des noms (restriction d'influence des noms).

9.4 Critère "mobilite"

9.4.1 Nature des entités migrantes

Les entités migrantes sont les processus du calcul. Plus généralement, on pourrait considérer que tout message (i.e. un message au sens du π-calcul, ou bien un processus migrant, ou encore une instruction de contrôle de l'état d'une localité) qui figure dans une configuration est une entité migrante. Toutefois, seuls les processus sont l'objet d'instructions de migration explicites. Une localité en particulier ne peut migrer.

Dans le premier article mentionné ci-dessus, seuls sont autorisés à migrer des processus qui ne contiennent pas de récepteur public (mais on peut installer à distance un récepteur sur un canal nouveau). Ainsi, un récepteur public dans une configuration a une localisation fixe. Dans le second article, cette restriction est levée, et divers modèles de la migration sont discutés, correspondant à diverses restrictions du modèle général (toujours dans le cadre d'un calcul avec récepteur globalement unique). La nécessité d'avoir un mécanisme de "suivi" des récepteurs et d'acheminement des messages est évoquée, mais non traitée. C'est une part importante de ce que devrait faire l'implémentation d'un tel modèle, qui, en présence de pannes, n'est pas évidente.

9.4.2 Expression de la migration

La migration s'exprime sous la forme d'une instruction (processus élémentaire) $\mathsf{spawn}(a,P)$ par laquelle le processus P est envoyé à la localité a pour s'y exécuter. C'est donc une forme objective de mobilité, qui est asynchrone (mais on peut évidemment en programmer une version synchrone par la technique usuelle du π-calcul, avec canal de retour). Dans cette construction le processus P est passif, et ne commence son exécution qu'une fois parvenu à destination. La migration a donc aussi un caractère de synchronisation. L'espace des domaines étant plat, et les domaines étant directement "accessibles" les uns depuis les autres, il n'y a pas de topologie à gérer.

Le premier article sur le calcul discute l'utilité d'une telle construction – qui n'est pas claire, dans le cadre d'un calcul où la communication est globale et transparente. Toutefois, un argument qui n'est pas évoqué concerne les défaillances: un processus peut quitter un site qui va tomber en panne, et il n'est pas évident qu'on puisse simuler cela correctement avec la seule communication (problème d'atomicité).

9.4.3 Mobilité en cours d'exécution

Du fait que la migration s'exprime par une instruction du langage, la mobilité en cours d'exécution est possible: il suffit d'imbriquer ces instructions $\mathsf{spawn}(a,P)$. On contrôle exactement ce qui peut migrer, puisque, comme on l'a vu, la migration a un caractère bloquant.

9.4.4 Migration d'un état associé

Dans $\pi_{1\ell}$, comme dans tous les modèles fondés sur le π-calcul, il n'y a pas de notion d'état associé à un processus (quoique l'on puisse considérer l'ensemble des récepteurs connus d'un processus comme constituant son état). Lorsqu'on exécute une instruction $\mathsf{spawn}(a,P)$, seul le processus P se déplace donc vers la localité a. Toutefois il faut noter que le calcul utilise les définitions récursives de processus (plutôt que la replication), dont la sémantique procède par substitution. Par conséquent si un terme $\mathsf{spawn}(a,P)$ se trouve englobé dans une définition récursive, ce qui migre n'est pas simplement P, mais le processus obtenu en substituant dans P les variables récursives par leur définition. Un processus récursif ayant des paramètres, ceux-ci peuvent être considérés comme définissant un état.

9.4.5 Entretien de références distantes

Le processus migrant P dans $\mathsf{spawn}(a,P)$ conserve tous les liens qu'il peut avoir avec des récepteurs (ou des émetteurs) sur les noms de canaux qu'il connait. En effet la simple connaissance d'un nom suffit à pouvoir établir une communication distante. Il n'y a pas de système de délégation (proxies) à installer – au niveau du "programmeur", c'est à dire du langage –, puisque l'acheminement des messages est transparent. Le contexte du migrant est donc fixe, mais sans rupture des liens.

Chapitre 10

Seal calcul

10.1 Références bibliographiques

J. Vitek, G. Castagna, Seal: A Framework for Secure Mobile Computations. In Internet Programming Languages, 1999.
J. Vitek, G. Castagna, Towards a Calculus of Secure Mobile Computations. In Workshop on Internet Programming Languages, Chicago, Illinois, 1998.
J. Vitek, G. Castagna, Mobile Computations and Hostile Hosts. In Proceedings of the 10th JFLA (Journées Francophones des Langages Applicatifs), Avoriaz, France, January 1999.

10.2 Présentation générale

Le Seal Calcul est un calcul de la distribution "à grande échelle", conçu dans le but d'exprimer les propriétés essentielles de programmes Internet. Ce calcul est construit sur la base du π-calcul, et privilégie la mobilité de code et le contrôle d'accès aux ressources. Il est bâti sur cinq principes de conception: l'absence d'état global, l'explicitation des localités, la connectivité restreinte, la configuration dynamique, et le contrôle d'accès.

10.3 Critère "distribution"

10.3.1 Entités de base

Le Seal calcul définit trois abstractions: les processus, les ressources et les domaines. Les domaines capturent à la fois le concept de frontière physique et celui de frontière logique. Les processus capturent le concept de flot de contrôle (thread ou processus système), et se présentent sous forme d'une succession de capacités à communiquer et à migrer. Les ressources sont supposée être aussi bien des ressources physiques (zone mémoire, périphériques) que des services de niveau applicatif, de niveau système ou de niveau runtime. Toutefois dans le calcul, les seules ressources considérées sont les canaux de communication, qui sont utilisés, comme dans le π-calcul, pour synchroniser des processus concurrents et communiquer des noms. Ils sont nommés et localisés.

10.3.2 Manifestations des domaines

Dans le Seal calcul, un domaine est un "seal", qui est une localité contenant des processus s'exécutant, et d'autres "seals". Comme dans les Ambients, des domaines de même nom peuvent coexister.

10.3.3 Noms/références

Les entités nommées sont les canaux et les domaines ("seals"). Plusieurs domaines distincts peuvent porter le même nom, et un même canal peut avoir plusieurs recépteurs différents. La portée des noms est, comme dans le π-calcul, définie par l'opérateur de restriction.

10.3.4 Concept de domaine

Topologie de l'espace des domaines

Les seals sont structurés hiérarchiquement. Par voie de conséquence, un seal s'identifie à l'arbre des sous-localités qu'il recouvre. La topologie de l'arborescence des domaines est évolutive, car il y a mobilité de domaines.

Sémantique des domaines

Un seal abrite des ressources (les canaux), et gère l'accès à ces ressources. De ce point de vue, un seal est un domaine de sécurité. Un seal est aussi une unité de migration, au sens ou le calcul prévoit la migration atomique d'un seal.

10.3.5 Communication locale

Désignation locale

La désignation d'un canal par un processus hébergé dans le même seal que le canal[1] s'effectue au moyen du nom du canal, décoré par un symbole spécial indiquant le caractère local de la désignation. La désignation locale est donc explicite dans le calcul.

Politique de communication locale

Il s'agit de la communication entre deux processus d'un même domaine, sur un canal local. Ce type de communication est synchrone.

Sémantique de la réception

La réception est multiple, comme dans le π-calcul ou Nomadic Pict.

10.3.6 Communication distante

Désignation distante

La désignation distante (désignation d'un canal hébergé par un seal autre que le seal du processus désignant) n'a de sens que depuis un seal parent (père ou fils). Le mode de désignation est entièrement conditionné par la relation entre les habitats respectifs des entités désignantes et désignées (explicitation de la localisation relative). Plus précisément, la désignation d'un canal hébergé dans un seal n par un processus hébergé dans un seal fils de n s'effectue au moyen du nom du canal, décoré par un symbole spécial indiquant le lien de parenté avec n. La désignation du même canal, par un processus hébergé par le seal père de n, s'effectue par nom du canal décoré par n.

Politique de communication distante

Un processus père et un processus fils (i.e. hébergés dans deux localités mère et fille respectivement) peuvent communiquer de façon synchrone (up-sync ou down-sync). La communication distante est limitée à la communication entre père et fils, sur un canal situé sur le seal père ou sur le seal fils. De plus, pour que la communication distante ait bien lieu, il faut que l'autorisation en soit explicitement donnée, sous la forme d'un capacité, par le seal hébergeant le canal de communication utilisé.

Routage logique

Du fait que seule la communication père-fils est autorisée, le routage des messages dans l'arborescence des seals doit être programmé explicitement.

1. le calcul parle dans ce cas de canal *local*.

10.4 Critère "mobilité"

10.4.1 Nature des entités migrantes

Les entités migrantes sont les domaines, c'est à dire les seals. Il y a également une autre forme de migration, qui consiste à envoyer un message dans une localité parente.

10.4.2 Expression de la migration

La migration s'exprime comme un envoi de message: le fait d'envoyer le nom d'un seal sur un canal a pour effet de déplacer un seal (s'il en existe) de ce nom voisin du message (i.e. en parallèle) dans la localité où se situe la communication, c'est à dire soit la localité du message, soit dans le seal père, soit chez un fils. Comme pour la communication, ceci est contrôlé explicitement par un "portail" qui ouvre l'accès au canal. Il faut noter que la migration dans le seal n'est pas un simple déplacement: on peut en effet créer plusieurs copies (éventuellement aucune, auquel cas on a une destruction du seal concerné) du seal migrant, sous des noms différents. L'instruction de migration est objective – elle s'exerce de l'extérieur du seal déplacé – et asynchrone, le seal déplacé s'exécutant indépendamment de l'ordre de migration.

10.4.3 Mobilité en cours d'exécution

On vient de voir que la migration s'effectue indépendamment de l'exécution de ce qui migre. On peut donc dire qu'elle a lieu "en cours d'exécution".

10.4.4 Migration d'un état associé

Dans le Seal calcul il n'y a pas de notion d'état. Le "contexte" d'une entité migrante reste toujours fixe.

10.4.5 Entretien de références distantes

Le seal migrant conserve la connaissance des noms de canaux sur lesquels il possède une référence, mais on ne peut pas considérer qu'il s'agit de références distantes. En effet ces noms désignent (éventuellement) des entités du contexte d'arrivée du migrant, et non celles auxquelles ils faisaient référence avant le déplacement. De plus, le seal migrant peut changer de nom après son déplacement.

10.5 Critère "sécurité"

Un des buts du seal calcul est de permettre de prouver des propriétés de sécurité pour des processus mobiles, le code de sécurité étant localisé dans de petits protocoles pouvant être prouvés indépendamment de leur contexte[2].

Le calcul envisage la sécurité en terme de protection des ressources d'une structure d'accueil contre des agents malveillants ainsi que celle des agents contre la structure d'accueil elle-même, cet aspect étant largement inexploré jusqu'ici.

Le seal calcul prend en compte les attaques suivantes: divulgation ou modification non autorisée d'informations, déni de service, ou introduction de chevaux de Troie dans un système:

- Le premier type d'attaque est pris en compte par la portée lexicale et l'utilisation de capacités linéaires pour protéger de manière forte les ressources.
- La prise en compte d'attaques de déni de service fait l'objet de recherches en cours. Le calcul doit être étendu en élargissant la notion de ressource à de la mémoire ou du temps cpu.
- Le calcul se prémunit également contre les chevaux de Troie: des seals ne peuvent pas se dissimuler à l'intérieur d'un seal accrédité à pénétrer dans un système et en sortir une fois l'accès autorisé pour endommager des ressources situées à l'intérieur du système (pas de "hitch-hiking"). La mise en place d'un processus en parallèle avec ceux contenus dans un seal pour interférer avec la manière dont s'effectue la réception est également interdite.

2. Une théorie de la preuve reste encore à élaborer pour définir rigoureusement les notions d'observation, de test et de spécification.

Cependant:
- Des attaques de rejeu sont possibles car le calcul autorise la duplication de seals [3].
- Malgré la protection offerte par le calcul, une structure d'accueil peut falsifier les informations qu'elle fournit à des seals, suivre l'itinéraire d'un seal particulier, le piéger, usurper son identité, ou espionner les communications entre différents seals.

10.5.1 Identification et authentification

Les seals sont identifiés par leurs noms. Comme en spi-calcul, il est possible de décrire des protocoles d'authentification en seal.

10.5.2 Contrôle d'accès

Politique d'accès aux ressources

Le calcul ne modélise pas de politique de sécurité [4] mais fournit des mécanismes de protection permettant de mettre en oeuvre des politiques de sécurité flexibles.

Modèle d'autorisation

Les mécanismes de protection portent sur les aspects suivants:
- **Contrôle de l'utilisation des noms:** il est assuré par la portée lexicale qui protège en terme de confidentialité et d'intégrité les noms de canaux qui sont des capacités à communiquer i.e. les processus ne peuvent utiliser que les canaux dont ils connaissent les noms.
- **Contrôle de l'utilisation des ressources:** des capacités linéaires [5], révocables appelées **portails** sont utilisées pour contrôler de manière fine l'accès aux ressources.
- **Contrôle de la migration et de la communication:** le modèle de protection du seal calcul est hiérarchique: un seal ne peut migrer qu'avec la permission explicite de son seal parent. Il ne peut monter ou descendre que d'un seul niveau dans la hiérarchie de seals.
  A l'aide du mécanisme de portails, la communication inter-seals est restreinte, en étant placée sous le contrôle du seal parent: un seal ne peut communiquer qu'avec son parent ou avec l'un de ses fils.

Domaine de sécurité

La frontière d'un seal peut représenter un domaine de protection e.g. un pare-feu ou un "bac à sable". Elle ne peut pas être dissoute contrairement au calcul des ambients. Un processus qui s'exécute à l'intérieur d'un seal est toujours protégé par cette frontière: il ne peut pas s'échapper dans l'environnement pour agir de manière malveillante et ce dernier ne peut pas accéder à aux ressources contenues dans le seal sans une permission explicite.

Le calcul garantit la propriété dite du "pare-feu idéal" i.e. $(\nu x)x[P] \simeq \mathbf{0}$ car si le nom d'un seal est inconnu de tout autre processus, il ne peut pas y avoir de portail ouvert pour ce nom, ou de communication sur un canal contenu dans ce seal.

10.5.3 Sécurisation des communications

On dispose d'une première protection au niveau langage à l'aide de l'opérateur de restriction et de la portée lexicale comme en spi-calcul. [6] La confidentialité d'un message peut être atteinte si l'environnement est capable de deviner les noms libres à l'intérieur d'un seal.

Une extension du calcul à l'aide de primitives cryptographiques comme en spi-calcul est envisagée.

3. Bien qu'introduisant une faiblesse du point de vue sécurité, la duplication a été permise dans le calcul pour permettre à des applications de mettre en oeuvre de la réplication e.g. dans un but de tolérance aux défaillances.

4. Une extension du calcul en ce sens est en cours de développement.

5. Les capacités sont linéaires au sens où elles accordent à un seal donné le droit d'utiliser un portail en émision ou en réception une fois seulement. Les portails protègent un seal contre les seals qu'il peut contenir ainsi que contre son parent.

6. Une clé cryptographique peut être représentée ainsi: si z est le texte en clair et x la clé de chiffrement, alors $K_x(z)$ peut se représenter par le seal $y[!\overline{x}^{\uparrow}(z)]$.

Troisième partie

Annexe et Conclusion

Annexe A

Fiche d'analyse type

A.1 Références bibliographiques

A.2 Présentation générale

A.3 Critère "distribution"

A.3.1 Entités de base pour la distribution

Entités de base

Manifestations de domaines

Noms/références

A.3.2 Concept de domaine

Topologie de l'espace des domaines

Sémantique des domaines

Observabilité/contrôlabilité des domaines

A.3.3 Communication locale

Désignation locale

Politique de communication locale

Sémantique de la réception

A.3.4 Communication distante

Désignation distante

Politique de communication distante

Routage logique

A.3.5 Domaines et défaillances

A.4 Critère "mobilité"

A.4.1 Nature des entités migrantes

A.4.2 Expression de la migration

A.4.3 Mobilité en cours d'exécution

A.4.4 Migration d'un état associé

A.4.5 Entretien de références distantes

A.5 Critère "sécurité"

A.5.1 Identification et authentification

A.5.2 Contrôle d'accès

Politique d'accès aux ressources

Ressources et granularité du contrôle d'accès

Privilèges et attributs de contrôle

Modèle d'autorisation

Délégation de privilèges

Domaine de sécurité

A.5.3 Sécurisation des communications

A.5.4 Audit

A.5.5 Administration d'une politique de sécurité

Conclusion

Il est instructif de dresser un tableau comparatif des langages et modèles étudiés, selon les critères que nous avons retenu. Ce tableau étant une version à deux dimensions de la série de fiches que nous avons présentée, nous ne le dressons pas intégralement ici, mais on pourrait en tirer plusieurs constatations:

► en ce qui concerne la *sécurité*, qui est évidemment un aspect important s'agissant de code mobile, on voit que, mis à part le cas d'Agent Tcl, presque rien n'est fait dans les modèles étudiés. Le spi et le sjoin-calcul fournissent une première approche, en ce qui concerne la sécurisation des communications, mais ne sont pas encore implémentés. D'un autre côté, un certain nombre de techniques sont connues (voir le chapitre 3), et on peut se poser la question de savoir lesquelles il serait bon d'intégrer à un modèle ou un langage de programmation pour la mobilité.

► sur les "entités de base", les modes de communication et la nature de entité migrantes, les divers modèles et langages diffèrent largement, sauf bien entendu ceux qui sont basés sur le π-calcul (Dπ, JOIN, Nomadic Pict, $\pi_{1\ell}$). On ne peut toutefois en tirer de conclusion particulière en ce qui concerne la mobilité, puisque cette diversité reflète celle des langages de programmation sous-jacents. À l'inverse, le critère "sémantique des domaines" s'avère, du moins par rapport à l'échantillon examiné, très "classifiant": on constate certes que les domaines sont toujours des "unités d'exécution"[1], mais ils peuvent aussi être – ou non – "unité de communication". Dire qu'un domaine est unité de communication signifie qu'il n'y a pas de communication directe possible d'un domaine à un autre: il faut faire migrer explicitement les messages, par l'intermédiaire d'un "proxy" par exemple. On parle alors de communication *locale*, confinée à l'intérieur d'un domaine, par opposition à la communication *globale* si le domaine n'est pas l'unité de communication[2].

► un autre point sur lequel on voit s'établir un clivage net entre les divers modèles est fourni par le critère "topologie de l'espace des domaines": les domaines forment en effet une structure qui est soit "plate", non imbriquée, soit hiérarchique. Cette dichotomie correspond en fait, dans l'échantillon étudié, à une distinction que l'on peut faire concernant la nature des entités migrantes: dans les modèles à structure hiérarchique, l'entité migrante est le "domaine" (ambient, localité...), tandis que dans un espace "plat" de domaines, ce sont les "processus" (programmes, clôtures...) qui migrent. De même, si l'on excepte le cas du Seal, cette distinction coïncide avec celle que l'on peut faire concernant l'expression de la migration: la migration peut être "objective", imprimée "de l'extérieur" de l'entité migrante, ou au contraire "subjective". Cette distinction va de pair avec le caractère synchrone ou asynchrone de la migration, par rapport à l'exécution de l'entité migrante: la migration (objective) de programme, est bloquante, tandis que dans la migration (subjective ou objective) de "domaine", le moment où la migration s'effectue est indépendant a priori de ce qui se passe dans le migrant; en particulier, des messages peuvent y arriver de façon asynchrone par rapport à la migration. Typiquement, pour employer les notations de $\pi_{1\ell}$ et des Ambients par exemple, dans $\mathsf{spawn}(s,P)$ le processus P est "gelé" jusqu'à son arrivée dans le site s, tandis que dans $u[\mathsf{in}\, v.P \mid Q]$ le processus Q peut s'éxecuter, et en particulier recevoir des messages, sans que soit fixé le moment où cette exécution est interrompue par la migration dans "l'ambient" u.

Si l'on s'en tient aux caractéristiques qui, relativement à la migration, opèrent des clivages significatifs, on peut donc résumer – très grossièrement! – l'analyse que nous avons faite dans le tableau suivant, où nous

1. cet aspect "unité d'exécution" ne prend véritablement son sens que dans les modèles où la défaillance est prise en compte, c'est à dire dans $\pi_{1\ell}$ et le JOIN (calcul).

2. l'absence de communication globale n'est pas contradictoire avec le fait qu'un nom peut avoir – ce qui est le cas dans l'ensemble des modèles étudiés – une portée globale. Techniquement, on a portée globale lorsqu'il y a une possibilité "d'extrusion" de la portée d'un nom au delà d'un domaine. Cela s'écrit, en utilisant par exemple la notation des ambients: $u[(\nu\, v)P] \equiv (\nu\, v)u[P]$.

identifions la dichotomie entre migration "bloquante" et migration "asynchrone" avec celle qui est établie (dans l'échantillon considéré) entre migration "de programme" et migration "de domaine"[3]:

langage/modèle	communication		migration	
	locale	globale	bloquante	asynchrone
Agent Tcl, Seal	×			×
Ambients, Nomadic Pict	×			×
Dπ, Kali Scheme	×		×	
JOIN		×		×
Obliq, $\pi_{1\ell}$		×	×	

Dans Agent Tcl, la communication est globale, mais non "transparente", dans la mesure où le nom de l'agent avec lequel on veut communiquer contient aussi son adresse, i.e. le nom du site dans lequel cet agent s'exécute. D'où la position intermédiaire de ce langage en ce qui concerne la communication. Une remarque similaire vaut pour le Seal, où la communication n'est pas strictement locale, mais limitée aux parents proches. On voit que toutes les autres possibilités sont représentées. Un avantage que peut présenter un langage fondé sur la communication globale est que le programmeur se trouve dans un modèle de haut niveau, avec une vision qui fait abstraction de la répartition effective des "ressources" auxquelles il envoie des requètes. D'un autre côté, le modèle, de plus bas niveau, de la communication locale peut donner plus de possibilités de contrôle (gestion des échecs par exemple); il est de toute façon à la base de la vision plus abstraite de la communication transparente (voir Nomadic Pict).

La migration "subjective", asynchrone, de domaine, donne directement une modélisation de l'idée d'agent à la Telescript (Nomadic Pict par exemple en est une formalisation assez directe), mais offre moins de possibilité de contrôle sur l'état de l'entité migrante, puisque la sémantique laisse indéterminé l'ordre temporel entre la réception de message par le "domaine" en voie de migration et son déplacement effectif. Il faut remarquer d'ailleurs que très souvent (dans la plupart des exemples de programmes présentés en illustration de la migration), la migration "subjective" est utilisée de façon quasi "objective", le contenu du domaine migrant se limitant à une instruction de migration suivie d'une continuation, se trouvant éventuellement dans un environnement de "définitions" (*cf.* le JOIN). D'un autre côté, dans la migration "objective", bloquante, de programme (excepté donc la migration du Seal), on a un contrôle strict sur ce qui migre[4]; on peut toujours écrire du code migrant en cours d'exécution, mais la modélisation des agents est moins claire. La migration de domaine parait plus puissante, mais cela n'est vrai que si des messages peuvent être émis depuis un domaine vers l'extérieur, donc en l'absence de pannes. De plus, la migration de domaine semble plus lourde à implémenter. Il serait donc peut-être souhaitable d'avoir les deux formes dans un langage.

Il reste une "dimension" selon laquelle les diverses propositions étudiées se séparent, qui est la "gestion du contexte" de l'entité migrante. Nous avons abordé ce point avec les critères "migration d'un état associé" et "entretien de références distantes". On voit bien que les langages – Agent Tcl, Kali Scheme, Obliq – mettent un accent particulier sur ce point, mais il est difficile d'en tirer une conclusion précise car, à l'exception d'Obliq peut-être, la sémantique des langages est très imprécise (*cf.* la fiche Kali Scheme par exemple). D'un autre côté, la notion "d'état" est absente des calculs que nous avons étudié, et l'on peut donc se demander si ceux-ci permettraient de formaliser, ou du moins de modéliser la sémantique – c'est à dire l'implémentation – des langages pour la mobilité, en rendant compte par exemple des différentes manières de "gérer le contexte" que nous avons évoquées au chapitre 2.

Enfin une dernière conclusion générale que l'on peut tirer de cette étude est que, bien que le concept de domaine nous ait paru important de différents points de vue (communication, défaillance, sécurité, administration...) il reste, sur le plan théorique et plus encore dans les langages, fort peu développé et exploité.

3. qui coïncide aussi, de manière plus contingente, avec la distinction "espace plat/hiérarchique" de domaines, et, à l'exception du Seal, avec la distinction "migration objective/subjective".

4. ce n'est pas tout à fait clair dans le cas de Kali Scheme, car la détermination de ce qui migre exactement dépend fortement de l'implémentation.

Table des Matières

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399