Systèmes distribués : transparence, masquage et outils associés

Résumé : Ce document traite du problème, toujours actuel, de l'unification des paradigmes de programmation locale et de programmation distribuée. Après une définition de cette notion d'unification, nous présenterons cinq caractéristiques des systèmes distribués que nous considèrons comme fondamentales : les pannes partielles, la concurrence, la confiance, la mémoire répartie et la latence. Nous verrons comment ces caractéristiques peuvent être prises en charge de façon transparente dans un but d'unification ainsi que les contraintes imposées par une telle transparence.

Nous présenterons ensuite nos contributions dans le domaine de la prise en charge de la caractéristique de la mémoire répartie au travers de la bibliothèque de communication JToe. Par ailleurs, nous avons contribué à la prise en charge de la latence par l'introduction transparente d'asynchronisme dans une application orientée objets. Nous verrons les possibilités offertes par ce nouvel asynchronisme et nous prouverons, à l'aide du π-calcul, le respect de la sémantique séquentielle de l'application synchrone d'origine.

Ces travaux nous amènent à proposer un nouveau modèle original de programmation concurrente : les bouquets d'activations.

Discipline : Informatique

Mots-Clefs : Systèmes distribués, programmation concurrente, bouquets d'activations, unification, transparence, π-calcul, asynchronisme.

Distributed Systems : Transparency, Masking and Associated Tools

Abstract : This dissertation deals with the unification of the local and distributed computing models. After defining this concept of unification, we will describe five characteristics of distributed systems that we believe are essential : partial failures, concurrency, trust, distributed memory and latency. We will show how this characteristics can be managed transparently to provide the unification as well as the constraints imposed by this transparency.

Next, we will present our contributions in distributed memory management through the JToe library. Furthermore, we have contributed to latency management by introducing transparent asynchronism in object oriented applications. We will describe what this new aynchronism allows and prove, using π-calculus, that the sequential semantic of the original synchronous application is conserved.

This work leads us to propose a brand new concurrent programming model : bouquets of activations.

Discipline : Computer-Science

Keywords : Distribued systems, concurrent programming, bouquets of activations, unification, transparency, π-calculus, asynchronisme.

LaBRI,
Université Bordeaux 1,
351 cours de la Libération,
33405 Talence Cedex (FRANCE).

N° d'ordre : 3074

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Pascal Grange**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Systèmes distribués : transparence, masquage et outils associés

---

Soutenue le : 2 décembre 2005

Après avis des rapporteurs :

Denis Caromel . . Professeur
Jean-Louis Pazat . Professeur

Devant la commission d'examen composée de :

Raymond Namyst	Professeur	Président
Denis Caromel . .	Professeur	Rapporteur
Serge Chaumette	Professeur	Examinateur
Jean-Louis Pazat .	Professeur	Rapporteur

– 2005 –

A la mémoire d'Isabelle Attali, de Hugo et de Tom Caromel.

Résumé

Ce document traite du problème, toujours actuel, de l'unification des paradigmes de programmation locale et de programmation distribuée. Après une définition de cette notion d'unification, nous présenterons cinq caractéristiques des systèmes distribués que nous considèrons comme fondamentales : les pannes partielles, la concurrence, la confiance, la mémoire répartie et la latence. Nous verrons comment ces caractéristiques peuvent être prises en charge de façon transparente dans un but d'unification ainsi que les contraintes imposées par une telle transparence.

Nous présenterons ensuite nos contributions dans le domaine de la prise en charge de la caractéristique de la mémoire répartie au travers de la bibliothèque de communication JToe. Par ailleurs, nous avons contribué à la prise en charge de la latence par l'introduction transparente d'asynchronisme dans une application orientée objets. Nous verrons les possibilités offertes par ce nouvel asynchronisme et nous prouverons, à l'aide du π-calcul, le respect de la sémantique séquentielle de l'application synchrone d'origine.

Ces travaux nous amènent à proposer un nouveau modèle original de programmation concurrente : les bouquets d'activations.

Mots-Clefs : Systèmes distribués, programmation concurrente, bouquets d'activations, unification, transparence, π-calcul, asynchronisme.

Abstract

This dissertation deals with the unification of the local and distributed computing models. After defining this concept of unification, we will describe five characteristics of distributed systems that we believe are essential : partial failures, concurrency, trust, distributed memory and latency. We will show how this characteristics can be managed transparently to provide the unification as well as the constraints imposed by this transparency.

Next, we will present our contributions in distributed memory management through the JToe library. Furthermore, we have contributed to latency management by introducing transparent asynchronism in object oriented applications. We will describe what this new aynchronism allows and prove, using π-calculus, that the sequential semantic of the original synchronous application is conserved.

This work leads us to propose a brand new concurrent programming model : bouquets of activations.

Keywords : Distribued systems, concurrent programming, bouquets of activations, unification, transparency, π-calculus, asynchronisme.

Table des matières

Table des figures

Faux départ

De tout temps et à jamais, l'homme a toujours distribué les tâches qu'il devait accomplir. . .

On assiste depuis quelques années à la multiplication des réseaux de toutes sortes. Cela soulève naturellement la question de l'exploitation de ces réseaux dans un contexte distribué. . .

Pour satisfaire l'accroissement de la demande en terme de capacité de calcul. . .

L'augmentation des puissances de calcul mais aussi des capacité de communication des machines modernes. . .

Aujourd'hui, rares sont les ordinateurs isolés. S'appuyant sur cette observation, de nombreux projets tentent de permettre l'exploitation . . .

Un grand nombre d'applications de calcul scientifique nécessitent des ressources de calcul et de mémoire sans cesse croissantes. . .

. . . Pas mieux . . .

Introduction

DNS, WEB, LDAP, NFS, Seti@Home, IMAP... autant de technologies et d'applications distribuées connues par le plus grand nombre et pour certaines par le grand public. Les applications distribuées constituent le cœur de l'informatique moderne telle qu'on la connaît depuis l'ère Internet. On désigne communément sous le terme de *système distribué* le ou les outils qui permettent le développement et/ou le déploiement de telles applications. Mais de quoi s'agit-il ?

On trouve dans la littérature différentes définitions, parfois contradictoires, de ce qu'est un système distribué. Coulouris, Dollimore et Kindberg dans *Distributed Systems : Concepts and Design* [33] considèrent qu'il s'agit d'un ensemble de composants situés sur des machines connectées par un réseau et qui utilisent l'envoi de messages comme seul moyen de coordination et de communication. Le *World Wide Web*, qui correspond à cette définition, n'est pourtant pas considéré comme un système distribué par Tanenbaum et Van Steen dans *Distributed Systems : Principles and Paradigms* [125]. Pour eux, un système distribué doit apparaître auprès de ses utilisateurs comme un système cohérent unique. Or les utilisateurs du Web voient bien que les documents qu'ils consultent sont situés à différents endroits et sont gérés par différents serveurs. Ce n'est donc pas à leurs yeux un système distribué.

Ces deux exemples montrent qu'aucune définition ne fait l'unanimité. Il existe cependant un certain nombre de particularités typiques des systèmes distribués. Waldo, Wyant, Wollrath et Kendall, dans l'article *A Note on Distributed Computing* [143] proposent quatre caractéristiques fondamentales des systèmes distribués qui les différencient des systèmes que nous dirons *locaux* :

- la *concurrence* ;
- les *pannes partielles* ;
- la *mémoire répartie* ;
- la *latence*.

La prise en charge de ces caractéristiques lors du développement d'une application nécessite la présence de constructions spécifiques au sein du paradigme de programmation. Cela induit des différences au niveau des paradigmes de programmation entre paradigmes de programmation *locale* et de programmation *distribuée*. Ces caractéristiques posant un certain nombre de contraintes au niveau du développement, de nombreux travaux ont

pour objectif de proposer leur prise en charge transparente. Waldo, Wyant, Wollrath et Kendall [143] parlent alors d'*unification* : une application écrite selon un paradigme de programmation *unifié* pourrait s'exécuter indifféremment sur un système *local* ou distribué.

Se pose alors la question de la faisabilité d'une telle unification. Selon eux, ces quatre caractéristiques empêchent l'unification du paradigme de programmation locale avec le paradigme de programmation distribuée. D'un autre côté, Spiegel dans *Automatic Distribution of Object-Oriented Programs* [117] explique pourquoi, de son point de vue, les arguments des auteurs précédents ne sont pas fondés. Il considère donc la question de l'unification des programmations locale et distribuée comme toujours ouverte.

L'objectif de cette thèse est de se pencher sur cette question de l'unification et comment, à cette fin, gérer de façon transparente les caractéristiques d'un système distribué. Nous décrirons ces caractéristiques et les pistes envisageables pour leur prise en charge transparente. Nous verrons les compromis qu'une telle transparence impose aux applications. Nous présenterons également nos contributions dans le domaine de la prise en charge de la confiance, de la mémoire répartie et de la latence. Avant cela, nous allons d'abord définir plus précisément ce que nous entendons par *unification*.

Unification des paradigmes de programmation locale et distribuée

La notion d'*unification* n'est jamais explicitement définie même si Waldo et al. [143] évoquent une fusion des paradigmes de communication et de programmation tout en refusant de complexifier la programmation locale. Quant à Spiegel [117], il pense que l'unification consiste à rendre un système distribué transparent.

Pour clarifier cela et fournir une définition de l'unification, nous considérons la notion de paradigme de programmation et celle de modèle d'exécution. Le paradigme de programmation est défini par l'ensemble des éléments caractéristiques du langage alors que le modèle d'exécution représente le contexte dans lequel un programme s'exécute.

On peut avoir un modèle d'exécution locale ou distribuée. Il semble clair qu'un modèle d'exécution distribuée implique l'utilisation d'un ensemble de machines distinctes, connectées entre elles par un réseau. Un modèle d'exécution locale, au contraire, est caractérisé par l'utilisation d'une seule machine.

La frontière entre paradigme de programmation locale et paradigme de programmation distribuée est plus floue. Nous considérerons qu'un *paradigme de programmation distribuée* fournit des constructions permettant l'interaction de plusieurs processus distincts grâce à un mécanisme de communication plus ou moins explicite et propose des outils de gestion des caractéristiques d'un système distribué. Nous reviendrons sur celles-ci

dans la prochaine section. Un *paradigme de programmation locale* peut se définir en opposition au paradigme de programmation distribuée en ce qu'il ne fournit pas les constructions permettant l'interaction entre plusieurs processus mais seulement la programmation d'un processus unique dans un espace mémoire unique. Cependant, nous considérerons que les paradigmes locaux intègrent les mécanismes aujourd'hui communs que sont les processus légers et les mécanismes de gestion de la mémoire et de la concurrence qui vont de pair.

Au départ, un paradigme de programmation locale est conçu pour un modèle d'exécution locale et un paradigme de programmation distribuée pour un modèle d'exécution distribuée. L'unification consiste à proposer un paradigme de programmation permettant le développement de programmes capables de s'exécuter naturellement selon les deux modèles. Si l'on s'interdit de complexifier le paradigme de programmation locale alors on peut définir l'*unification de la programmation locale et distribuée* de façon utopique comme la possibilité de réaliser des programmes selon un paradigme de programmation locale qui puissent s'exécuter de façon optimale quelque soit le modèle choisi, d'exécution locale ou distribuée. Autrement dit, les caractéristiques spécifiques aux systèmes distribués sont masquées au niveau du langage et leur gestion est reportée au niveau de la plate-forme d'exécution. Nous désignerons une telle plate-forme comme un système distribué transparent.

Une telle unification transforme toutes les applications existantes développées grâce à un paradigme de programmation locale en applications potentiellement distribuées. Celles-ci deviennent ainsi capables d'exploiter un ensemble de machines distinctes pour mener à bien leur exécution. Ceci présente deux intérêts : (1) l'augmentation *gratuite* des performances des applications ; (2) la rentabilisation d'un parc de machines existant via leur exploitation dans ce cadre distribué.

Cette unification présente cependant une limite triviale : un système distribué transparent peut difficilement être conçu à l'aide d'un autre système distribué transparent. De façon générale, ce sont toutes les applications intrinséquement distribuées qui ne sont pas adaptées pour une telle unification. Pour ces dernières, il est nécessaire d'avoir accès aux mécanismes spécifiques du distribué et leur écriture ne peut se limiter à l'exploitation de paradigmes de programmation locaux.

Ainsi, selon le type d'application considéré, il est possible d'aboutir à des conclusions différentes quant à la faisabilité d'une telle unification. Waldo et al. s'intéressant à des applications intrinsèquement distribuées, concluent naturellement que cette unification est impossible. Spiegel, au contraire, se penche sur l'exécution d'applications développées selon un paradigme de programmation locale sur un modèle d'exécution distribuée. Cette cible opposée explique son désaccord.

Caractéristiques des systèmes distribués

Selon Waldo et al. [143], un système distribué possède quatre caractéristiques fondamentales que nous avons déjà évoquées. Nous en ajoutons une cinquième qui nous paraît pertinente et sur laquelle nous reviendrons plus tard : la *confiance*. Nous allons maintenant décrire brièvement chacune d'entre elles. L'ordre dans lequel nous les présentons ne reflète aucune importance ou hiérarchie entre ces caractéristiques. Toutes constituent un obstacle ou une difficulté à l'unification dont nous venons de parler. Pourtant, on peut remarquer qu'elles ne sont pas toutes de même nature. Certaines sont directement liées à des phénomènes physiques, d'autres sont constitutives des systèmes distribués ou sont une conséquence de la distribution. Nous n'avons pas la prétention de croire qu'elles suffisent totalement à définir un système distribué. Cependant, le choix de ces caractéristiques nous a paru pertinent pour l'étude de l'unification et elles se retrouvent d'ailleurs dans la plupart des travaux autour des systèmes distribués. Sans limiter ces travaux à l'étude d'une seule caractéristique, on peut quand même citer, pêle-mêle : [20, 24, 64, 66, 95, 135] pour la latence, [5, 6, 7, 45, 75, 116, 119, 124, 133, 147] pour la mémoire répartie, [10, 21, 39, 42, 57, 82, 86, 131] pour les pannes partielles, [16, 31, 81, 98] pour la concurrence ou encore [27, 53, 113] pour la confiance. Nous reviendrons sur ces différents travaux lorsque nous entrerons plus en détail dans l'étude de ces caractéristiques au chapitre suivant.

La latence correspond à un phénomène physique. Le transport d'une information d'un nœud A vers un nœud B sur un réseau nécessite le déplacement d'éléments appartenant au monde physique : électrons, ondes radio, photons. La vitesse de ces déplacements est bornée par la vitesse de la lumière. En pratique, elle est également bornée par l'organisation et l'architecture du réseau : temps de traitement par des routeurs intermédiaires, temps de négociation du média lorsqu'il est partagé par plusieurs machines, etc. Le délai induit par l'ensemble de ces phénomènes physiques et/ou techniques forme ce qu'on appelle la latence. La latence est une caractéristique intrinsèque d'un système distribué de par l'éloignement physique ou topologique des machines mais nous verrons qu'on la retrouve également dans les environnements d'exécution locaux.

La latence étant une caractéristique liée à des phénomènes physiques ou matériels, elle ne peut ni être masquée, ni être réduite. On peut faire une brève comparaison avec la notion de bande passante, qui représente la quantité de données qui peut transiter sur un support pendant une période donnée. Si la bande passante entre deux machines est trop faible par rapport à l'application visée, il est toujours possible de l'augmenter en ajoutant un lien supplémentaire. Pour autant, la latence de ce lien n'aura pas changé. Nous verrons donc comment nous accommoder de cette latence et comment en limiter l'impact.

La mémoire répartie est une caractéristique constitutive des systèmes distribués. Chaque machine participant à un système distribué possède sa propre mémoire, son propre espace d'adressage et éventuellement ses propres périphériques de stockage permanents. La mémoire de l'ensemble du système est donc répartie dans le sens où une partie des données se trouve sur un site, une autre partie sur un autre et certaines données peuvent

éventuellement être dupliquées. La prise en charge de cette répartition implique la mise en œuvre de mécanismes de communication pour l'échange de données entre les différents sites. Cet échange peut être explicite, par l'utilisation d'une bibliothèque de communication. Il peut être abstrait dans la notion d'appel de routine à distance. Il peut enfin être implicite dans le cas d'une mémoire virtuellement partagée. Nous verrons les différents mécanismes de communication nécessaires à la gestion de cette mémoire répartie. Nous insisterons particulièrement sur les bibliothèques de communication au chapitre 2 et nous présenterons notre contribution dans le domaine du transfert d'objets en Java au chapitre 3.

Les pannes partielles peuvent être considérées comme une conséquence de la distribution. Le fait même qu'un ensemble de machines soit utilisé implique que ces dernières puissent tomber en panne indépendamment les unes des autres. Cette notion de panne partielle s'associe à celle de *fonctionnement partiel* qui peut être vue comme un atout par rapport à un système local. En effet, si un service est hébergé sur une seule machine, la panne de celle-ci implique l'arrêt du service. Si on imagine qu'il est déployé sur un ensemble de machines distinctes, il devient alors possible de profiter de la notion de panne partielle, ou plutôt de *fonctionnement* partiel, pour maintenir le service même dans le cas où l'une des machines ne fonctionne plus. Nous aborderons les pannes partielles dans les systèmes distribués ainsi que leur prise en charge transparente et les contraintes que cela impose.

La concurrence correspond au fait qu'un ensemble de machines indépendantes tentent d'accéder simultanément à une même ressource ou à un ensemble de ressources partagées. Il est alors nécessaire de conserver cette ou ces ressources dans un état cohérent. Nous étudierons deux mécanismes permettant cela. Nous verrons que le problème de concurrence est déjà présent dans les environnements d'exécution locaux à processus légers, ce qui permet de le gérer de façon transparente dans un environnement d'exécution distribuée. Nous étudierons les contraintes qu'il est alors nécessaire d'imposer pour assurer le respect de la cohérence de l'application d'origine.

La confiance entre les différentes machines d'un système distribué est un élément essentiel. Nous ajoutons donc cette caractéristique aux quatre précédentes. Dès lors que l'ensemble des machines participant à un système distribué ne sont pas toutes gérées et administrées par la même organisation, des problèmes de confiance entre ces machines apparaissent. Il est nécessaire de proposer des mécanismes pour gérer cette confiance et assurer une certaine protection en son absence. Nous nous intéresserons particulièrement au déploiement de code et aux problèmes spécifiques que cela peut soulever sur ce plan. Dans ce domaine, nous avons participé à l'élaboration d'une plate-forme à base de cartes à puce Java que nous présenterons section 1.3.2 page 23.

La prise en charge transparente des cinq caractéristiques décrites ci-dessus est nécessaire si on souhaite unifier les paradigmes de programmation locale et distribuée conformément à la définition de l'unification que nous avons donnée précédemment. Comme on peut le voir figure 1 page suivante, tant que le paradigme de programmation est adapté au modèle d'exécution, le rôle d'une plate-forme d'exécution se limite à

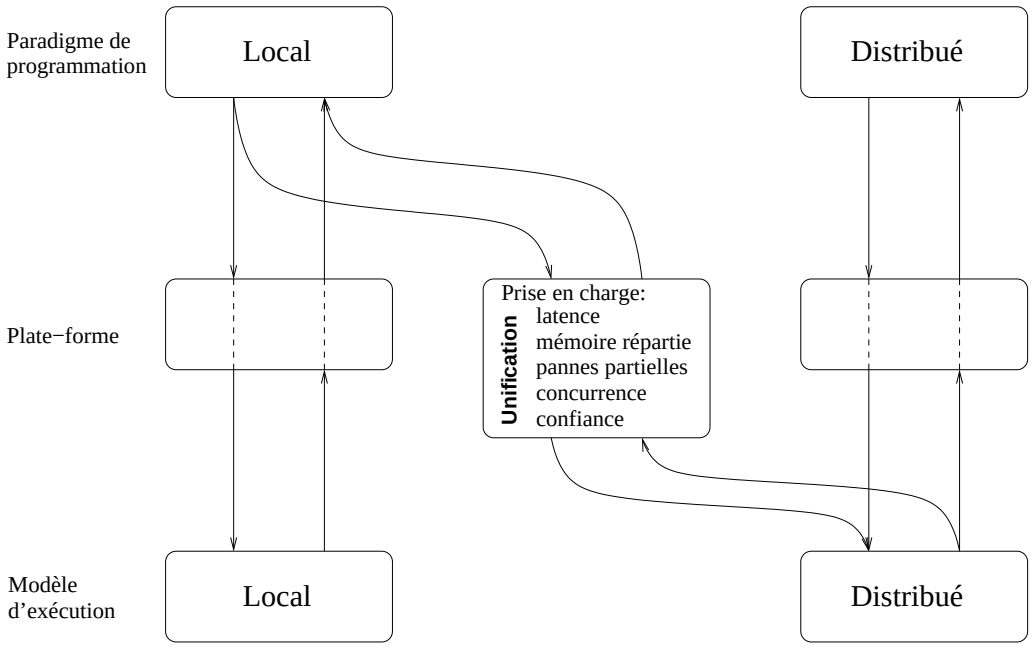


FIG. 1 – Unification : la plate-forme doit prendre en charge toutes les caractéristiques

faire l'interface entre ces deux éléments. Dans le cas de l'unification, c'est-à-dire si on souhaite exécuter un programme écrit selon un paradigme de programmation locale dans un modèle d'exécution distribuée, c'est à la plate-forme de prendre en charge toutes les caractéristiques que nous venons de décrire. Nous allons étudier au chapitre 1 chacune d'entre elles, comment les prendre en charge de façon transparente ainsi que les conséquences de leur masquage éventuel.

Organisation du document

Nos principales contributions dans le cadre de cette thèse se situent au niveau de la prise en charge de la mémoire répartie et de la latence qui sont traitées, respectivement, dans les parties I et II de ce document. Néanmoins, dans un soucis de cohérence, il nous a paru pertinent de traiter chacune des cinq caractéristiques que nous avons énumérées. Ce traitement fait l'objet du chapitre suivant. Nous y reviendrons plus en détail sur ce que ces caractéristiques peuvent recouvrir ainsi que sur les outils qui permettent de les gérer et nous verrons également ce qu'il est possible de faire en terme de masquage pour chacune d'entre elles.

Les caractéristiques nous ayant les plus particulièrement intéréssées au cours de cette thèse étant la mémoire répartie et la latence, nous reviendrons donc plus en détail sur

celles-ci pour la gestion desquelles nous avons proposé une bibliothèque de transfert d'objets et étudié une propriété pour l'ajout automatique d'asynchronisme dans une application objets.

Nous reviendrons plus en profondeur sur certains aspects de la prise en charge de la mémoire répartie au chapitre 2. Nous nous intéresserons en particulier aux messages actifs et aux appels de routine à distance. Dans ce dernier cas, nous présenterons une plate-forme qui masque la sémantique des appels distants derrière une syntaxe locale et les difficultés que cela introduit.

Nous proposons la bibliothèque JToe qui évite ce problème. Elle explicite la notion de communication tout en fournissant l'abstraction de l'envoi d'objets. Nous présenterons JToe au chapitre 3 en justifiant son intérêt et nous décrirons un exemple d'implantation ainsi que des mesures de performances.

Nous détaillerons la prise en charge de la latence par les mécanismes d'appel asynchrone de routine à distance au chapitre 4. Nous verrons les problèmes que peut présenter le masquage de l'asynchronisme.

Enfin, au chapitre 5 nous proposerons des propriétés sur les programmes permettant d'introduire de l'asynchronisme. Elles sont nommées propriétés de *séparabilité* et d'*activabilité*. Celles-ci sont des extensions de travaux effectués dans le cadre de la plate-forme ProActive [65]. Nous montrerons les modifications qu'elles autorisent sur un programme et nous prouverons leur validité. Nous aborderons également les possibilités en ce qui concerne leur détection dans les programmes et nous introduirons, comme perspective de nos travaux, la notion de *bouquet d'activations*.

En conclusion nous développerons une réflexion sur nos travaux et sur l'enjeu du masquage dans les systèmes distribués.

Bonne lecture.

Chapitre 1

Caractéristiques des systèmes distribués

Ce chapitre reprend les cinq caractéristiques des systèmes distribués que nous avons définies dans l'introduction :

- les pannes partielles ;
- la concurrence ;
- la confiance ;
- la mémoire répartie ;
- la latence.

Pour chacune d'entre elles, nous décrivons les mécanismes classiques de leur prise en charge puis nous examinons les possibilités de rendre celle-ci transparente. Nous revenons sur cette transparence à la section 1.6 page 43.

La mémoire répartie et la latence concernent directement les travaux que nous présentons, respectivement chapitres 3 et 5. Nous reviendrons plus en profondeur sur certains aspects de ces deux caractéristiques dans les chapitres 2 et 4.

1.1 Prise en charge des pannes partielles

Un système distribué exploite un ensemble de machines distinctes qui peuvent tomber en panne indépendamment les unes des autres. Nous allons nous intéresser ici aux outils permettant de gérer ces pannes partielles afin d'améliorer la fiabilité ou la disponibilité d'une application en profitant du *fonctionnement partiel* induit par les pannes partielles. Dans la suite de cette section, nous utiliserons le terme *processus* pour désigner de façon générique une machine, un service ou tout autre élément constitutif d'un système distribué qui peut être soumis à une panne.

Les pannes dans un système distribué peuvent être de différente nature et affecter divers composants. Une panne peut être franche, transitoire, intermittente ou byzantine. Une panne franche correspondra généralement à un *crash*, c'est-à-dire que le processus

cessera de répondre à toute requête ou d'émettre des requêtes. Dans le cas d'une panne transitoire ou intermittente, au contraire, la panne apparaîtra à un instant donné puis le processus fonctionnera à nouveau correctement. Enfin, les pannes byzantines sont caractérisées par un comportement *déviant* du processus affecté : ce dernier ne respecte plus sa spécification, soit à cause d'une opération malveillante, soit en raison d'une erreur logicielle ou encore à la suite d'une erreur physique non détectée comme une erreur mémoire. Quel que soit le type de panne qui apparaisse, une première difficulté consiste à la détecter avant de pouvoir la gérer.

1.1.1 Détection des pannes

En général, un mécanisme de détection de panne dans un système distribué décrète un délai – *time out* – au delà duquel, si aucune réponse n'a été reçue, le processus est considéré comme étant en panne. Dans un système synchrone pour lequel le délai maximum de communication et de traitement d'une requête est connu, ce système détectera les pannes correctement. Cependant, dans un système totalement asynchrone et pour lequel les délais de réponse ne sont pas prévisibles, comme c'est le cas pour la plupart des systèmes distribués, cette méthode ne permettra pas de distinguer un processus en panne d'un processus très lent. Ce phénomène abouti à l'impossibilité de la résolution de certains problèmes en présence d'une panne, notamment celui du consensus [50] dans lequel un ensemble de processus doivent se mettre d'accord sur une valeur commune. Afin de contourner cette impossibilité, Chandra et Toueg [21] ont proposé la définition de détecteur de panne non fiable. Dans leur modèle, chaque site possède son propre détecteur de panne. Ce détecteur est non fiable, c'est-à-dire qu'un processus peut être considéré en panne (suspecté) alors qu'il fonctionne encore. Des processus peuvent être ajoutés ou supprimés de la liste des processus suspectés d'être en panne au fur et à mesure du temps et cette liste peut être différente sur chacun des sites. Chandra et Toueg proposent une catégorisation des détecteurs de panne qui permet d'analyser les problèmes qui peuvent être résolus ou non en fonction du type de détecteur disponible et du nombre maximum de pannes possibles.

On voit apparaître ici une notion de subjectivité dans la panne. En effet, un détecteur de panne peut considérer un processus en panne tandis qu'un autre considère que ce même processus fonctionne. Chandra et Toueg s'intéressent à un modèle dans lequel tous les processus sont reliés entre eux deux à deux par un réseau fiable et ne prennent en compte que les pannes franches. La détection de pannes byzantines complique encore les choses, sans compter les pannes réseau. Dans ce dernier cas, deux processus peuvent mutuellement se détecter en panne alors que chacun continue à fonctionner mais que le réseau qui les relie est lui-même en panne ou simplement n'existe plus. C'est par exemple le cas avec des périphériques mobiles : le déplacement relatif de deux périphériques interrompt leur connexion – distance trop importante ou obstacles à la transmission dans le cas de réseaux sans fil – sans qu'aucun des deux ne soit en panne. En ce qui concerne la mobilité, on ne parlera d'ailleurs plus de panne puisqu'il s'agit d'une situation normale.

Nous allons nous intéresser maintenant à la façon d'exploiter une architecture distribuée pour éviter les pannes.

1.1.2 Gestion des pannes partielles

Il est possible d'exploiter les pannes partielles – ou plus exactement le *fonctionnement* partiel – afin d'augmenter la fiabilité d'un système distribué. Cela peut se faire principalement selon deux mécanismes, la réplication et la sauvegarde de points de reprise.

Dans le cas de la réplication, plusieurs copies d'un même processus sont organisées de telle façon que si l'un d'entre eux tombe en panne, un autre puisse prendre la relève. Le choix de l'emplacement des réplicas est essentiel. En effet, la panne d'une machine peut être compensée par l'utilisation d'une autre mais dans le cas d'une panne réseau, si tous les réplicas se situent sur le même sous-réseau, il est probable qu'aucun ne sera plus joignable. Il est également possible qu'une panne soit la conséquence d'un phénomène environnemental : tempête, inondation, incendie, etc. Il est donc intéressant de placer les réplicas dans des zones géographiquement et topologiquement distinctes [42]. La redondance peut se faire selon un mode actif ou passif. Dans ce dernier cas, plusieurs réplicas sont disponibles mais un seul est effectivement utilisé. Lors d'une défaillance de celui-ci, un réplica passif est sélectionné pour le remplacer. Dans le cas de la redondance active, toutes les requêtes sont transmises à tous les réplicas qui traitent chacune d'entre elles. Cependant, la réponse d'un seul de ces réplicas doit être fournie, ce qui implique de la sélectionner. Cette sélection peut se faire soit d'un commun accord entre les réplicas par un mécanisme d'élection – seul le réplica élu transmet ses réponses –, soit par l'utilisation d'un composant de gestion de la redondance chargé de filtrer les réponses et de sélectionner celles à retransmettre.

La sauvegarde de points de reprise va permettre, en cas de panne, de reprendre l'exécution d'un ensemble de processus à partir d'un point de reprise précédemment sauvegardé. La panne d'un ou plusieurs processus provoque l'arrêt de tous les processus, leur retour dans un état globalement cohérent précédemment enregistré, puis le redémarrage, à partir de cet état global, de l'ensemble des processus. La difficulté consiste ici à définir la notion d'état globalement cohérent [22]. En effet, chaque processus enregistre son propre état indépendamment des autres. En cas de panne, un processus est remis dans un état correspondant à un point de sauvegarde qu'il aura enregistré. Un état globalement cohérent assurera que les états des différents processus sont cohérents les uns avec les autres. Par exemple, si un processus $p1$ a transmis un message à un processus $p2$, après une panne et la restauration des processus, leurs états doivent être cohérents. C'est-à-dire que, soit $p1$ n'a pas encore transmis le message et $p2$ ne l'a pas reçu, soit $p1$ a transmis le message mais $p2$ ne l'a pas encore reçu. Mais $p2$ ne doit pas avoir déjà reçu le message si $p1$ est dans un état qui précède cet envoi de message.

Quelle que soit la technique de gestion de panne choisie – redondance active, passive, points de reprise –, il est nécessaire de dupliquer, d'une façon ou d'une autre, l'état des différents processus. Dans le cas des points de reprise on enregistrera ces états ainsi que les messages entre processus sur un support de stockage. Pour la réplication, il faudra maintenir une cohérence entre les états des différents réplicas, en assurant un ordre de

réception identique des requêtes pour la réplication active ou en effectuant des synchronisations régulières entre les différents réplicas et le réplica principal pour la réplication passive. Les contraintes de cohérence peuvent être relâchées selon la sémantique précise de l'application. Le service DNS [94], par exemple, utilise un mode de réplication passif. Pour synchroniser leur état, les serveurs DNS secondaires contrôlent, à intervalle de temps régulier, le numéro de version des tables du serveur primaire. Si ce numéro a été incrémenté, les serveurs secondaires se mettent à jour auprès du primaire. Le mécanisme permettant le maintien de la cohérence entre les états des serveur primaire et secondaires est ainsi peu coûteux, de par la fréquence réduite de ces mises à jour et de par l'exploitation de cette propriété au niveau du protocole.

Ces méthodes de gestion de panne ne suppriment pas totalement la possibilité d'une panne ingérable. Ces outils permettent *seulement* d'augmenter la fiabilité d'un système. En particulier, l'exploitation de ces mécanismes dans un système distribué peut contribuer à fournir une fiabilité plus importante que dans un système local. Nous revenons sur leur exploitation transparente dans la prochaine section.

1.1.3 Conclusion : prise en charge transparente des pannes partielles

On considère généralement dans un modèle d'exécution locale, qu'une panne est totale, c'est-à-dire que l'ensemble du système tombe en panne. L'exemple le plus simple est celui d'une coupure électrique. La notion de panne partielle semble donc spécifique aux systèmes distribués, ou du moins est elle mise en exergue par la fréquence importante à laquelle ce type de panne peut se produire.

La prise en charge transparente des pannes partielles peut se faire sous deux angles différents. On peut soit exploiter de façon transparente le *fonctionnement* partiel du système pour augmenter la fiabilité de l'application originelle, soit transformer une panne partielle en une panne totale, notion déjà présente – au moins implicitement – dans un paradigme de programmation locale.

Considérons une application écrite selon un paradigme de programmation locale. Son exécution dans un modèle d'exécution distribuée nécessite, entre autres, son *découpage* par la plate-forme d'exécution en n processus P_1 à P_n qui s'exécuteront sur les différentes machines disponibles. Chaque machine est équipée d'un détecteur de panne non fiable tel que toute machine en panne est suspectée par le détecteur de panne d'au moins une machine en fonctionnement au bout d'un temps fini. Lorsqu'un détecteur de panne suspecte une machine d'être en panne, il oblige sa propre machine à se mettre en panne, ce qui a pour effet, entre autres, d'interrompre le ou les processus en cours d'exécution sur cette machine. On peut se convaincre facilement qu'un tel dispositif garantit que, dés qu'une machine tombe en panne, toutes les autres machines seront en panne au bout d'un temps fini. De plus, considérons les états e_1 à e_n des processus P_1 à P_n au moment de la panne, volontaire ou accidentelle, de la machine sur laquelle ils se trouvaient. On se convainc aisément qu'il existe une exécution locale de l'ensemble des processus P_1 à P_n telle que,

à un instant de cette exécution, ces processus se trouvent dans ces mêmes états e_1 à e_n. Si on débranche notre machine à cet instant précis, provoquant, de fait, une panne totale, on se retrouve dans une situation similaire à celle provoquée par le dispositif de gestion des pannes partielles que nous venons de décrire. S'il existe un état e de l'application locale d'origine qui correspond à l'état global e_1 à e_n des processus alors ce mécanisme de gestion des pannes partielles est bien simulé par la notion de panne totale. C'est ce que doit garantir la transformation mise en œuvre par la plate-forme.

Ce mécanisme gère bien les pannes partielles de façon transparente mais au lieu d'augmenter la fiabilité par rapport à un environnement d'exécution local, on la diminue de façon importante puisque la panne d'une seule machine parmi n provoque la panne totale du système. Si la probabilité de panne d'un composant est relativement forte, les applications risquent de devoir recommencer leur exécution plusieurs fois avant de pouvoir se terminer sans panne, voire même n'auront jamais le temps de se terminer.[1]

Pour limiter ce problème, on peut envisager la mise en place transparente d'une méthode de gestion de panne. RPC-V [39], par exemple, utilise la réplication pour fournir un mécanisme d'appel de procédures à distance tolérant les pannes. Cependant, dans ce cas, seules les procédures sans effet de bord sont gérées, ce qui limite la transparence puisque l'utilisateur devra s'assurer que les procédures sont bien sans effet de bord avant de pouvoir l'exploiter. ProActive [10] propose l'utilisation transparente de points de reprise. L'état des différents processus de l'application (objets actifs) ainsi que les messages sont sauvegardés et permettent, après une panne, de restaurer l'application dans un état cohérent.

La gestion transparente des pannes pose certains problèmes. Par exemple dans le cas des points de reprise, si une application interagit avec l'extérieur – accès à un fichier, affichage d'un message, connexion à un serveur distant –, faut-il refaire ces interactions après une panne ou non ? De plus, sans connaître les spécificités de l'application, on est obligé de faire des hypothèses pessimistes en ce qui concerne la sauvegarde des points de reprise ou le maintient de la cohérence entre réplicas. Dans le cas de la réplication active, par exemple, tous les messages seront retransmis à tous les réplicas dans le même ordre alors que certains ne sont peut-être que de simples consultations sans impact sur l'état du processus. Vogels, van Renesse et Birman [136] constatent d'ailleurs, dans le cadre du déploiement d'infrastructures proposant une réplication transparente, que cette transparence n'est quasiment jamais exploitée mais que des solutions spécifiques aux applications sont mises en place.

[1] On peut d'ailleurs noter qu'on retrouve ce problème dans certains paradigmes de programmation non locale. C'est le cas, par exemple, du standard MPI dans lequel la panne d'un processus au sein d'un groupe ne peut pas être gérée par l'application [52].

1.2 Prise en charge de la concurrence

Un ensemble de processus peuvent partager une ou plusieurs ressources et y accéder de façon concurrente. Le maintien d'un état cohérent de ces ressources nécessite la mise en œuvre de techniques d'exclusion mutuelle ou peut également se faire par *auto-protection*. Nous allons étudier ces deux approches puis nous verrons comment la concurrence peut être gérée de façon transparente dans un système distribué.

1.2.1 L'exclusion mutuelle en distribué

Différents mécanismes existent pour garantir l'exclusion mutuelle entre processus dans le cadre de la programmation locale, comme les sémaphores [38] ou les moniteurs [69]. Les parties de programme accédant à la ou les ressources partagées et ne devant être exécutées que par un seul processus à la fois sont appelées *sections critiques*. Chaque section critique est protégée par une structure de données qui dépend du mécanisme d'exclusion mutuelle utilisé et que nous appellerons, de façon générique, *verrou*. Avant de commencer l'exécution d'une section critique protégée par un verrou v, un processus s'assure qu'aucun autre n'est en train d'exécuter une section critique protégée par ce même verrou v. Si ce n'est pas le cas, le processus attend que v redevienne disponible. C'est le mécanisme d'exclusion mutuelle mis en œuvre qui assure que ce fonctionnement est possible en exploitant généralement des primitives de bas niveau de la machine.

Dans le cas d'un modèle d'exécution distribuée, le mécanisme d'exclusion mutuelle poursuit les mêmes objectifs mais ne peut pas reposer sur des primitives de bas niveau. Il est nécessaire d'obtenir un consensus entre les différents processus par la mise en œuvre d'un algorithme distribué et, évidemment, par l'échange de messages entre ces processus. On peut distinguer principalement deux familles d'algorithmes distribués d'exclusion mutuelle : les algorithmes à base de permission et ceux à base de jeton. L'unification de ces deux familles étant représentée par les algorithmes centralisés dans lesquels un seul processus a pour rôle de coordonner tous les autres [107].

Dans le cas des algorithmes à base de permission, avant de commencer la section critique protégée par le verrou v, un processus demande à tous les autres la permission de le faire puis attend que chacun ait répondu. Lorsqu'une telle demande parvient à un processus, soit celui-ci ne souhaite pas entrer dans une section critique associée à ce verrou et dans ce cas renvoi sa permission, soit il souhaite également entrer en section critique protégée par v et cela aboutit à un conflit. Ce type de conflit peut être réglé en associant une date d'émission à chaque message [81].

Dans le cas des algorithmes à base de jeton, un seul et unique jeton par verrou existe et permet au processus qui le possède d'entrer en section critique. Ce jeton est échangé entre les processus, soit de façon continue – le jeton est échangé en permanence entre les processus –, soit à la demande – le processus qui souhaite entrer en section critique demande

le jeton à celui qui le possède. Le fait qu'un seul jeton existe garantit que l'entrée en section critique n'est accordée qu'à un seul processus et peut éventuellement permettre à un processus d'entrer plusieurs fois consécutives en section critique tant qu'il possède le jeton. Dans le cas d'un mouvement continu du jeton, les processus sont organisés en anneau virtuel qui assure que le jeton passera régulièrement par chacun d'entre eux. Dans le cas du jeton à la demande, un processus doit connaître celui auquel il doit demander le jeton, par exemple dans [98], à l'état initial, tous les processus connaissent le propriétaire du jeton. Schématiquement, lorsqu'un processus veut le jeton, il adresse sa demande au dernier propriétaire dont il a eu connaissance. Ce dernier se charge de transmettre la requête au dernier propriétaire dont lui-même a eu connaissance puis considérera par la suite le processus demandeur comme le dernier propriétaire. Cet algorithme structure l'ensemble des processus comme un arbre, cet arbre évoluant en fonction des requêtes pour acquérir le jeton.

Ces mécanismes d'exclusion mutuelle reposent sur une collaboration entre les différents processus concernés. Dans le cas où un processus ne respecte pas l'algorithme, l'intégrité de la ressource n'est plus garantie. Leur sensibilité aux pannes partielles est importante. Dans le cas des algorithmes à base de jeton, une panne peut provoquer la perte du jeton et empêcher un processus d'entrer en section critique par la suite. Il est alors possible de mettre en œuvre un mécanisme de détection de perte de jeton et d'émission d'un nouveau jeton mais là encore d'autres problèmes doivent être réglés, notamment pour garantir l'unicité du jeton. De même pour les algorithmes à base de permission : la panne d'un des processus empêche les autres d'entrer en section critique. Pour éviter cela des mécanismes à base de quorum peuvent être mis en place : l'autorisation d'un sous ensemble des processus peut suffire à l'entrée en section critique.

Nous verrons section 1.2.3 page suivante que ces mécanismes d'exclusion mutuelle en distribué peuvent être utilisés pour implanter de façon transparente les primitives d'exclusion mutuelle déjà présentes dans un paradigme de programmation locale.

1.2.2 Mécanismes d'auto-protection

Les mécanismes d'auto-protection permettent au développeur de s'assurer que des accès multiples à une ressource donnée ne vont pas endommager cette dernière en protégeant directement la ressource elle-même. A l'inverse de l'exclusion mutuelle, les processus accédant à une ressource n'ont pas besoin de se coordonner, c'est la ressource qui va directement gérer les accès simultanés.

Ce type de mécanisme peut être mis en place directement au sein de la ressource via l'appel à des primitives de synchronisation. Dans ce cas, la ressource est accessible à plusieurs processus simultanément et seules les parties critiques de celle-ci sont protégées, généralement grâce à des mécanismes d'exclusion mutuelle locaux. Cela permet d'offrir un degré de concurrence fort dans l'accès à une ressource mais nécessite qu'elle ait été

prévue pour fonctionner dans un tel contexte. On parlera généralement de ressource *réentrante*.

Une autre approche consiste à encapsuler la ressource dans un processus chargé de gérer la cohérence des accès à cette dernière. Ces accès deviennent alors des requêtes auprès de ce processus qui se chargera des accès effectifs en s'assurant qu'ils maintiennent la ressource dans un état cohérent, par exemple en les séquentialisant. Des mécanismes de type objets actifs [20, 84] ou références asynchrones [135] sur lesquels nous reviendrons au chapitre 4 permettent d'assurer ce type de protection.

L'intérêt de l'auto-protection, comparé à l'exclusion mutuelle, est que l'accord de l'ensemble des processus n'est plus nécessaire. De plus, la panne d'une partie des processus accédant à la ressource ne peut ni corrompre la ressource ni empêcher les autres d'y accéder (sauf si la ressource elle-même tombe en panne). Cependant, la protection mise en œuvre ne concerne que *cette* ressource. Dans le cas où un ensemble d'opérations doit être effectué sur un ensemble de ressources distinctes, l'auto-protection ne suffit plus et des sections critiques doivent être mises en œuvre par des mécanismes d'exclusion mutuelle distribuée.

L'auto-protection permet d'encapsuler une ressource écrite selon un paradigme de programmation locale afin de la rendre accessible de façon transparente – pour la ressource elle-même – tout en assurant sa cohérence dans un modèle d'exécution distribuée.

1.2.3 Conclusion : prise en charge transparente de la concurrence

La programmation à base de processus légers est présente dans un paradigme de programmation locale. Ainsi, une application locale prend déjà en charge les problèmes d'accès concurrents à ses ressources par ses processus légers. Elle peut le faire soit grâce à l'exclusion mutuelle, soit en *auto-protégeant* une ressource.

L'exécution dans un modèle d'exécution distribuée d'une application locale nécessite son *découpage* par la plate-forme d'exécution en n processus communicants. Ce découpage peut directement se baser sur les processus légers présents dans l'application ou se faire selon une autre règle. Dans tous les cas, la plate-forme d'exécution simule les processus légers de l'application d'origine avec ce que nous appellerons des *processus virtuels*. Ainsi, toutes les opérations que l'application locale effectuait sur ses processus légers continuent de fonctionner sur les processus virtuels quelque soit l'organisation physique des processus générés par la plate-forme.

La prise en charge transparente de la concurrence consiste donc à implanter de façon distribuée les mécanismes d'exclusion mutuelle utilisés par l'application. De plus, une application développée selon un paradigme de programmation locale peut faire des hypothèses sur le nombre de processus légers pouvant accéder simultanément à une ressource. Cela oblige la plate-forme d'exécution à limiter l'accès aux ressources de l'application, y compris aux données de l'application éventuellement rendues accessibles à distance par la plate-forme, aux seuls processus virtuels de cette dernière. Autrement dit, un processus quelconque, n'ayant aucun lien avec l'application locale d'origine, ne doit pas pouvoir manipuler des données ou des ressources de cette application.

1.3 Prise en charge de la confiance

La coopération et les interactions entre composants gérés par des organismes indépendants, voire concurrents, posent de nombreux problèmes sur le plan de la sécurité et de la confiance entre ces différents composants. Les systèmes distribués sont concernés par l'ensemble des problèmes de sécurité des systèmes d'information. Dans le cadre de cette thèse, nous avons participé à la mise en place d'une infrastructure pour la sécurisation de codes mobiles et nous allons nous pencher sur ce point dans cette section.

La notion de code mobile soulève le problème de la confiance entre le fournisseur d'un code et l'entité d'exécution de ce code. Une confiance totale de l'un envers l'autre n'est possible que si, en réalité, les deux ne font qu'un. Dans le cas contraire, des mécanismes de protection doivent être mis en œuvre. En particulier, il est nécessaire que l'entité d'exécution puisse se protéger du code qu'elle exécute et inversement, que le fournisseur du code puisse se protéger de l'entité d'exécution.

La notion de code mobile est une notion large qui couvre différents paradigmes de programmation. On pourra se référer à [135, chap. 3] pour une présentation plus détaillée de ce concept et des paradigmes associés. Pour les besoins de cette section, nous retiendrons deux éléments qui, selon nous, sont caractéristiques des codes mobiles et qui permettent de mettre en évidence les problèmes de sécurité qui sont soulevés lorsque l'entité d'exécution et le fournisseur du code sont distincts :

1. il existe un intérêt, pour le fournisseur du code, et éventuellement pour l'entité d'exécution, à l'exécution de ce code et au résultat produit par cette exécution ;
2. l'obtention du code, son déploiement et son exécution sur l'entité d'exécution sont des opérations indissociables et leur enchaînement est automatique, aucun contrôle ni aucune validation intermédiaire par l'utilisateur n'est possible.

Nous considérons qu'un code mobile possède au moins une de ces deux caractéristiques. Celles-ci mettent clairement en évidence la question de la confiance entre l'entité d'exécution d'un code et le fournisseur de ce code. Dans la suite de cette section, nous considérerons toujours que l'entité d'exécution et le fournisseur du code sont distincts l'un de l'autre.

Lorsqu'un code mobile possède la caractéristique 2, il est indispensable de protéger l'entité d'exécution vis-à-vis du code mobile. Les *proxy* de la plate-forme Jini [142], par exemple, correspondent à ce type de code mobile : la communication entre un client et un service passe par un *proxy* dont le code est fourni par le service mais qui s'exécute sur le client et l'obtention, le déploiement et l'exécution du *proxy* sur la machine du client sont des opérations indissociables dont l'enchaînement est automatique (caractéristique 2). Le client doit donc être protégé du *proxy*. Nous revenons sur ce type de protection dans la prochaine section.

Lorsqu'un code mobile possède la caractéristique 1, il est indispensable de protéger le fournisseur du code de l'entité d'exécution. C'est le cas par exemple dans le projet Seti@Home [3] : le code de calcul fourni par Seti@Home s'exécute sur les machines

des participants mais les résultats produits par ces exécutions intéressent directement le projet Seti@Home (caractéristique 1 page précédente). Le projet Seti@Home doit donc se protéger des machines des participants qui pourraient, par exemple, fournir de faux résultats. Nous nous pencherons sur la protection du fournisseur du code section 1.3.2 page 21.

Dans le cadre de l'unification des paradigmes de programmation locale et distribuée, la plate-forme d'exécution est en charge de *découper* une application locale afin d'exécuter les processus générés sur différentes machines. Le résultat produit par ces exécutions intéresse directement le *fournisseur* de l'application locale d'origine. Dans le cas où, parmi les machines sur lesquelles les processus générés sont exécutés, certaines n'appartiennent pas au fournisseur du code, les problèmes de sécurité que nous évoquons ici doivent être pris en charge par la plate-forme d'exécution.

1.3.1 Protection de l'entité d'exécution

On dira d'un programme qu'il est *de confiance* s'il a les moyens de lire ou de détruire toutes les données présentes sur la machine sur laquelle il est installé. L'exécution de ce type de programme présente des risques difficiles à maîtriser. Les programmes de confiance doivent donc être sélectionnés rigoureusement et leur installation relève d'un procédé volontaire et conscient.

Dans le cas des codes mobiles, l'obtention, le déploiement et l'exécution peuvent constituer des opérations indissociables qui ne mettent pas forcément en œuvre la volonté ou l'accord explicite de l'utilisateur. Dans ce cas, ils ne doivent pas être *de confiance*. Il est donc nécessaire de limiter les opérations qu'ils sont autorisés à effectuer. Dans la suite, nous désignerons par le terme générique de *ressource* l'ensemble des éléments à protéger.

Le principe général permettant de limiter les opérations qu'un programme est susceptible d'effectuer repose sur le principe de machine virtuelle et en particulier de la virtualisation des ressources à protéger. Les programmes sont isolés des ressources par un ensemble de ressources virtuelles. Cette organisation est représentée figure 2 page suivante. Ces ressources virtuelles sont chargées de filtrer les accès aux ressources réelles en interdisant les accès non autorisés et en répercutant les accès légitimes. Par exemple si un code souhaite accéder à un fichier sur disque, la ressource virtuelle représentant le disque peut s'assurer que l'accès est bien autorisé avant de réaliser l'opération effective. La condition *sine qua non* pour que les règles de protection définies et contrôlées par les ressources virtuelles soient respectées, est que le contournement de ces ressources virtuelles soit impossible, ce qui est représenté par la courbe barrée sur la gauche de la figure 2 page ci-contre. Sur notre exemple précédent, si le code était capable d'accéder directement au disque dur, il pourrait contourner le contrôle de la ressource virtuelle. C'est le rôle de la machine virtuelle d'empêcher ces contournements et de permettre aux ressources virtuelles d'accéder aux ressources réelles correspondantes. Pour cela, ses deux principaux outils sont la protection de la mémoire et des modes d'exécution *privilégiés*

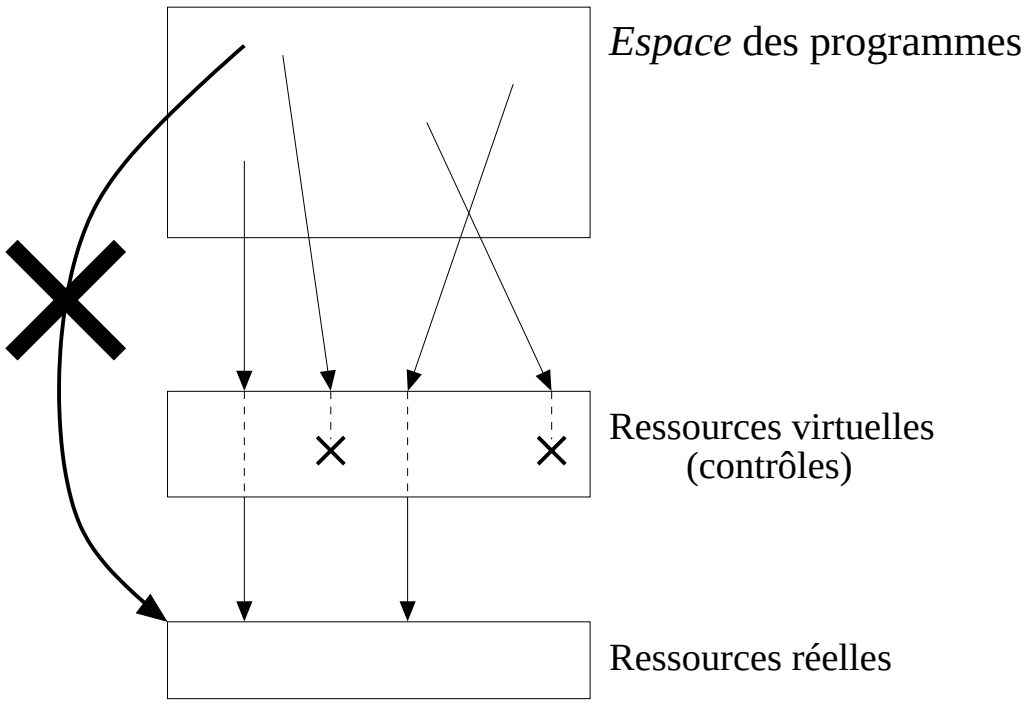


FIG. 2 – Protection des ressources.

pour les ressources virtuelles.

La protection assurée par une machine virtuelle peut reposer sur des outils matériels ou logiciels. On utilisera par la suite les termes *machine virtuelle matérielle* et *machine virtuelle logicielle* pour désigner chacune de ces approches. Les machines virtuelles matérielles sont représentées par la plupart des systèmes d'exploitation actuels. Parmi les machines virtuelles logicielles, on trouve les plate-formes Java [89] et .Net [88].

Afin d'accéder à une ressource protégée par le système d'exploitation, un processus doit déléguer les accès, généralement via des appels système, à une ressource virtuelle qui la représente et qui contrôle les accès de ce processus à cette ressource. Pour éviter que ce mécanisme puisse être contourné, le système d'exploitation utilise les mécanismes de mémoire virtuelle présents au niveau du matériel. Ainsi, toute la zone mémoire auquel un processus peut accéder est propre à ce processus, ce qui l'empêche de modifier ou de lire les zones mémoires d'autres processus et l'empêche également d'accéder aux zones mémoires des ressources réelles. Une ressource virtuelle s'exécutera dans un mode privilégié afin de pouvoir accéder effectivement à la ressource réelle.

Dans le cas d'une machine virtuelle logicielle, un certain nombre de contraintes est imposé sur le jeu d'instructions qu'elle peut exécuter. La protection se fait par la vérification que les codes qu'elle exécute respectent bien ces contraintes. Parmi celles-ci, qui ont principalement pour objectif la protection de la mémoire, on trouve l'interdiction d'effectuer de l'arithmétique des pointeurs ou de tenter d'accéder à des cellules de tableau inexistantes. Si l'une ou l'autre de ces opérations était autorisée, une machine virtuelle

logicielle ne pourrait plus garantir la protection de la mémoire.

La vérification d'un code s'effectue généralement en deux phases : au chargement du code dans la machine virtuelle, c'est-à-dire avant qu'il ne soit exécuté, puis tout au long de son exécution. Dans le cas de la plate-forme Java, les vérifications lors du chargement du code ont principalement pour objectif de s'assurer de sa conformité à la spécification. La plate-forme Java est orientée objets ; une classe définit un ensemble de champs et de méthodes que posséderont ses instances. Le code exécuté par une machine virtuelle Java est appelé *bytecode*. Les instructions du *bytecode* sont typées. Cela permet de vérifier, par exemple, qu'une référence n'est pas additionnée avec un entier, pour éviter l'arithmétique des pointeurs. Par ailleurs, la machine virtuelle s'assure qu'aucune classe ne peut manipuler des données auxquelles elle n'a pas accès (champs privés d'une autre classe, par exemple). Pour garantir cela, la machine virtuelle s'assure également de la compatibilité des types lors d'une affectation ou, à défaut, de la présence d'un test de validité (instruction *cast*) explicite dans le programme. Ce procédé de vérification assure qu'un certain nombre de principes de protection de la machine virtuelle ne sont pas violés par le *bytecode* chargé. C'est, entre autre, le typage du *bytecode* qui permet ces vérifications.

Il est également primordial qu'un programme ne puisse pas accéder à des cellules de tableau inexistantes. Cela lui permettrait de lire ou d'écrire une zone mémoire quelconque. Le respect des bornes des tableaux est vérifié tout au long de l'exécution du code.

La plate-forme .Net [88] fonctionne selon ces mêmes principes mais le langage bas niveau qu'elle exploite, l'*Intermediate Language*, propose des instructions permettant l'arithmétique des pointeurs. Ces instructions ne sont pas vérifiables et les parties de code y ayant recours doivent bénéficier d'une permission particulière pour pouvoir s'exécuter. Cela permet, par exemple, l'exécution de programmes C++ existants ayant recours à l'arithmétique des pointeurs.[2]

Les mécanismes que nous venons de décrire permettent de limiter les droits d'un code mobile. Ces droits sont basés sur une identité. Dans le cas d'un système d'exploitation, les droits sont généralement associés à l'identité d'un utilisateur. On devra donc faire en sorte d'exécuter le code mobile sous l'identité d'un utilisateur dont les droits correspondent à ceux qu'on veut attribuer à ce code mobile. Cela implique généralement une intervention manuelle de la part de l'administrateur de la machine, ce qui complique la mise en œuvre lorsque l'obtention, le déploiement et l'exécution du code mobile sont indissociables (caractéristique 2 page 17). De plus, l'interaction locale avec un code mobile sera relativement complexe dans la mesure où il s'exécutera dans son propre espace mémoire, comme un processus à part entière.

Dans le cas de la plate-forme Java, au contraire, les droits sont associés à l'identité de la classe dont le code est exécuté.[3] Cela permet de spécifier directement, pour un code

[2] On notera cependant que ces programmes ne pourront pas exploiter l'héritage multiple normalement présent en C++ mais absent de .Net.

[3] Il est également possible de limiter les permissions en fonction de l'identité d'un utilisateur en utilisant JAAS –Java Authentication and Authorization Service[121].

mobile donné, les droits qui lui sont accordés. De plus, l'ensemble des classes chargées au sein de la machine virtuelle s'exécutent dans le même espace mémoire et les communications se font directement par appel de méthode. Ainsi, différents codes, d'origines diverses, peuvent cohabiter au sein du même processus tout en étant protégés les uns des autres. La plate-forme Jini [142] utilise ce mécanisme ainsi qu'un système dynamique de permissions pour assurer la sécurité du client d'un service vis-à-vis du *proxy* de ce service. Le *proxy* d'un service Jini peut donc s'exécuter dans le même processus que le reste du code du client et il assure la communication avec le service en implantant le protocole nécessaire sans pour autant représenter une menace, le client étant protégé par la machine virtuelle et les permissions limitées (connexion au service) qu'il aura accordées au *proxy*.

Que l'on s'appuie sur la protection offerte par une machine virtuelle matérielle ou logicielle, cette protection est limitée par les failles qui peuvent se trouver au sein de cette machine virtuelle et qui pourraient être exploitées par un assaillant. On trouvera dans [37] une liste des failles qui ont pu être répertoriées dans les machines virtuelles Java. La sécurité des machines virtuelles logicielles reposant sur la validité du code exécuté, une opération de type arithmétique des pointeurs effectuée depuis l'extérieur de ce code ne sera ni constatée ni contrée. L'idée proposée dans [55] est alors que la modification d'un bit en mémoire par un rayonnement physique puisse affecter l'adresse contenue dans une référence. Cette référence pointerait alors vers une adresse aléatoire en mémoire, ce qui permettrait la mise en œuvre, par la suite, de mécanismes d'arithmétique de pointeurs. Govindavahala et Appel [55] proposent un programme augmentant la probabilité qu'une telle modification de la mémoire par une perturbation externe permette la réussite d'une attaque. Ils présentent des résultats expérimentaux obtenus en chauffant les composants mémoires à l'aide d'une lampe électrique.

Les mécanismes de sécurité que nous avons présentés peuvent paraître non transparents : lorsqu'un accès est refusé à un code mobile, cela doit lui être indiqué. Cependant, ils ne sont absolument pas spécifiques à la notion de code mobile et sont déjà tous pris en charge dans des paradigmes de programmation locale et ne nécessitent donc pas de construction particulière dans un paradigme de programmation distribuée. La protection de l'entité d'exécution peut donc être vue comme une notion transparente vis-à-vis du code mobile.

1.3.2 Protection du fournisseur

Nous venons de voir des mécanismes permettant à une entité d'exécution de se protéger des codes qu'elle exécute en limitant leurs droits. Ainsi on peut envisager l'exploitation des codes mobiles en limitant les risques de l'entité d'exécution hébergeant un tel code. Dans un certain nombre de cas, cela est suffisant. Jini [142] ou encore RMI [145], sur lequel nous reviendrons plus tard, utilisent les codes mobiles pour permettre l'interaction entre un client et un service. Le *proxy* fourni par le service est exécuté sur le client et se charge des communications avec le service. Il est donc indispensable que le client soit protégé de ce *proxy*. Le *proxy*, par contre, n'est pas protégé : le client peut perturber son

exécution et modifier son comportement. Les conséquences de ce type de malveillance seront limitées aux communications entre le *proxy* perturbé et le service. L'exécution du service lui-même ne peut pas être corrompue par le client, puisqu'il s'exécute sur une machine distincte et il suffira d'écrire un service robuste tolérant les requêtes émises par un *proxy* invalide.

Dans d'autres situations, le bon comportement du code mobile et les résultats produits par son exécution son vitaux. C'est le cas lorsque le fournisseur du code est intéressé par ces derniers (caractéristique 1 page 17). Le fournisseur devra alors pouvoir se protéger d'entités d'exécution susceptibles de corrompre l'exécution de son code. C'est le cas par exemple lorsqu'une application de calcul est déployée sur Internet : on doit pouvoir garantir que les résultats fournis par les différents codes s'exécutant sur les diverses entités d'exécution sont valides.

Il existe un nombre relativement important d'applications de calcul qui exploitent un ensemble de machines disséminées sur Internet, spécialement lorsque ces dernières sont peu ou pas utilisées. Nous parlerons alors de calcul opportuniste. Parmi ces applications, on peut citer le projet decrypthon [41] qui a pour objectif de faire progresser la recherche dans le domaine des maladies génétiques et des maladies rares et qui a déjà permis la comparaison de 550000 protéines entre mars et mai 2002 ou encore le projet de recherche d'intelligence extra-terrestre par l'écoute de l'univers Seti@Home [3]. Ce dernier projet repose aujourd'hui sur la plate-forme BOINC (Berkeley Open Infrastructure for Network Computing) qui supporte d'autres projets de calcul opportuniste [15]. Pour la plupart de ces projets, l'installation du code de calcul sur une machine requiert une intervention humaine[4] et volontaire. Cependant, la validité des résultats produits par l'exécution de ce code sur une machine distante est primordiale pour son fournisseur (caractéristique 2 page 17. Nous allons étudier maintenant les approches logicielles et matérielles qui peuvent être mises en œuvre pour assurer la protection du fournisseur du code vis-à-vis de l'entité d'exécution.

Approche logicielle

La seule méthode de protection logicielle généralement appliquée consiste à faire effectuer des calculs redondants par différents contributeurs puis à s'assurer que tous les résultats de ces calculs correspondent. Cela limite les risques qu'un résultat erroné soit considéré comme juste. Cette méthode est mise en œuvre dans le projet Seti@Home. On peut retrouver des propositions reposant sur la redondance des calculs pour garantir la validité d'un résultat dans [113].

Un autre enjeu dans le cas d'applications de calcul basées sur le volontariat est celui de la motivation des volontaires. Un élément fondamental de motivation est, bien entendu, le sujet du calcul lui-même, qui touchera les volontaires en fonction de leurs goûts personnels. Une autre source d'émulation consiste à recueillir des statistiques sur la contribution

[4] Pour ne pas dire *humaniste*.

des volontaires au calcul – temps de calcul consacré – puis à les exploiter pour la valorisation des participants, que ce soit par une publication en ligne des meilleurs contributeurs ou des concours entre équipes de contributeurs comme le fait le projet Seti@Home. Une autre possibilité est la rémunération des participants selon le même type de critère. Même dans le cas où les participants ne sont pas rémunérés, la tentation peut exister de tricher pour être bien classé parmi les autres participants. De telles pratiques ont d'ailleurs déjà été constatées avec le projet Seti@Home [3].

Pour éviter ce type de fraudes, Golle et Mirinov [53] proposent un mécanisme qui s'applique à un type précis de calculs dont le but est l'inversion de fonctions mathématiques à sens unique, c'est-à-dire, connaissant la fonction à sens unique f, connaissant y, trouver x tel que $f(x) = y$. La méthode consiste à calculer la fonction f sur tout son ensemble de définition jusqu'à obtenir y, on aura alors trouvé la bonne valeur de x. La distribution de ce type de calcul vers un ensemble de participants consiste alors à fournir à chaque participant la fonction f, la valeur y ainsi qu'un intervalle de l'ensemble de définition de f sur lequel ce participant devra travailler. Lorsqu'un participant a trouvé la valeur de x, il le signale, sinon, lorsqu'il a terminé le calcul sur l'intervalle qui lui était affecté, il en demande un nouveau et est crédité pour son travail. Une façon triviale de frauder est alors de demander régulièrement un nouvel intervalle sur lequel travailler en prétendant avoir terminé le travail sur le précédent tout en n'effectuant, en réalité, aucun calcul. Pour empêcher cela, une des idées proposées dans [53] est de choisir un ensemble de valeurs sentinelles $x_1 \dots x_n$ réparties uniformément sur l'intervalle de travail d'un participant et de calculer les valeur $y_1 \dots y_n$ de f pour ces points. On fournit ensuite à ce participant, en plus de f, de y et de l'intervalle de travail, l'ensemble des valeurs $y_1 \dots y_n$. Si le participant trouve x, il le signale mais il indique également lorsqu'il trouve une des valeurs sentinelles $x_1 \dots x_n$. Ainsi le travail correspondant à un intervalle sera crédité au participant uniquement s'il a bien trouvé l'ensemble des valeurs $x_1 \dots x_n$, ce qui garantit qu'il aura effectué une quantité minimale de travail. D'autres idées sont également proposées afin de compliquer encore la tâche d'un tricheur qui s'arrêterait de calculer dés qu'il aurait trouvé suffisamment de sentinelles [53].

Approche matérielle

Comme nous venons de le voir, des techniques logicielles permettent au fournisseur d'un code de se protéger d'une entité d'exécution déloyale, par exemple en rejetant les résultats produits par une exécution de son code perturbé par cette entité d'exécution. Pour cela, le code doit respecter certains critères : simple *proxy* ne servant qu'à communiquer vers un serveur robuste aux communications frauduleuses ; possibilité de comparer des résultats issus de plusieurs clients ; calcul d'une fonction ayant des propriétés mathématiques particulières.

Nous avons participé à l'élaboration d'une plate-forme à base de cartes à puce dans laquelle c'est le matériel qui garantit qu'aucune intervention extérieure ne pourra perturber l'exécution d'un code. Pour cela, nous exploitons les fonctionnalités des cartes à

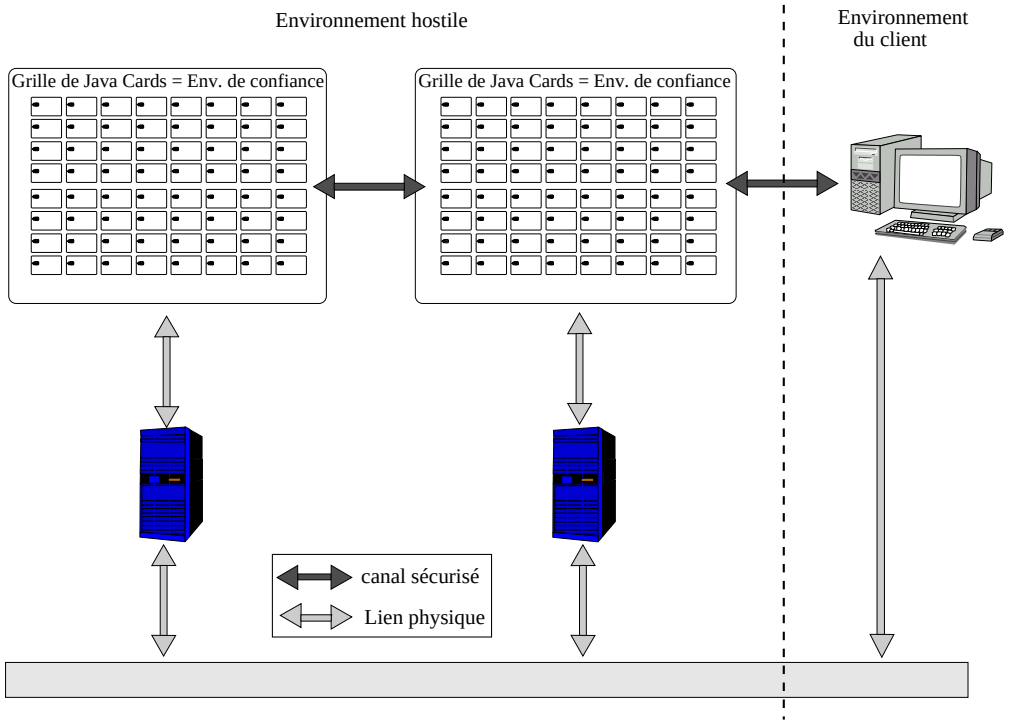


FIG. 3 – Infrastructure de calcul sécurisé sur une grille de cartes à puces.

puce multi-applicatives, et en particulier des *Java cards*, pour proposer et développer un environnement d'exécution distribué sur ces cartes à puce [27, 30].

Le principe de cette plate-forme est de rendre disponible, sur un site non sécurisé, un ensemble de cartes à puce multi-applicatives Java, ces cartes étant elles-même sécurisées. L'utilisateur de la plate-forme a la possibilité de charger dynamiquement un code Java-Card sur une ou plusieurs de ces cartes. Son code est alors préservé des atteintes extérieures par la conception même de la carte. Le transfert du code entre le client et la carte s'effectue de façon sécurisée grâce à des mécanismes cryptographiques. Ainsi, seule la carte cible, préalablement authentifiée comme telle auprès de l'utilisateur, peut déchiffrer le code transmis. Par la suite, ce code est installé dans la carte qui est la seule à y avoir accès. Même le propriétaire de la carte ayant un accès physique à celle-ci ne peut accéder aux données.[5]

Le code ajouté à la carte peut ensuite être utilisé à distance de façon sécurisée par le client et par lui seul. Ainsi notre plate-forme permet l'invocation de méthode à distance sur une carte à puce hébergée sur une machine et via un réseau qui ne sont pas de confiance, et cela de façon sécurisée. La figure 3 illustre ce fonctionnement.

Une application de calcul du fractal de Mandelbrot a été déployée sur cette infrastructure : le calcul du Mandelbrot est divisé en zones, chacune des zones est transmise

[5] Dans un délai et à l'aide de moyens *raisonnables*.

à différentes cartes à puce qui fournissent ensuite le résultat de leur calcul à l'application cliente, qui peut alors afficher le Mandelbrot en question. On pourra consulter une présentation détaillée de cette application dans [27]. L'ensemble des données transitant entre le client et les cartes sont sécurisées et même l'hôte accueillant les cartes ne peut les déchiffrer. Ceci nous garantit, d'une part, la confidentialité du code de calcul du Mandelbrot ainsi que du Mandelbrot calculé. D'autre part, le client peut avoir confiance dans les résultats produits, même à distance. Cette application nous a permis de démontrer la faisabilité d'un calcul distribué sur la grille sécurisée même dans le cas où les unités de calcul sont hébergées par une organisation qui n'est pas de confiance.

Bien entendu, en ce qui concerne le calcul distribué lui-même, cette infrastructure constitue une preuve de concept mais nécessite : (1) une architecture aussi sécurisée qu'une carte à puce mais dont les performances se rapprochent de celles d'un ordinateur classique ; (2) le déploiement, à une large échelle, de ce type d'architecture. Nous reviendrons dans la suite sur les questions éthiques qu'un tel déploiement peut soulever.

Au delà de l'approche purement *calcul* qui n'est pas forcément réaliste sur des cartes à puce actuelles, il apparaît qu'un certain nombre d'applications d'un genre relativement nouveau peut être envisagé grâce à cette infrastructure. Par exemple, des applications dans lesquelles le code d'une organisation A peut accéder aux données d'une organisation B sans que A ne divulgue son code à B ni B ses données à A. Une telle situation peut être illustrée par une relation entre Air-France et le FBI. On imagine que le FBI exige de pouvoir analyser la liste des passagers des vols Air-France à destination des USA mais que la compagnie aérienne souhaite protéger la confidentialité des données concernant ses passagers. [6] Dans ce cas, il est envisageable de déployer l'ensemble des données et du code sur une grille de cartes à puce. Le code de l'application FBI, chargé d'identifier les passagers suspects, est alors protégé par la carte. Les données d'Air-France sont partiellement accessibles et également protégées par la carte. Lorsque l'application FBI détecte un passager satisfaisant un ensemble de critères qui le rend suspect, elle peut alors transmettre sa clé au FBI (sans autre information), qui peut ensuite demander toutes les informations le concernant à Air-France, qui pourra les retrouver grâce à la clé fournie [74]. Cet environnement de confiance que représente la carte permet d'envisager de contrôler que l'application FBI ne retourne que des clés de passagers valides et aucune autre information. A cette condition on pourra lui donner accès à toutes les données contenues sur

[6] Cet exemple est inspiré d'une situation réelle de demande, par le *Bureau des douanes et de la protection des frontières* des USA, des dossiers commerciaux des compagnies aériennes européennes concernant les passagers sur les vols à destination des USA. Ces dossiers, généralement appelés PNR, pour *Passenger Name Record*, contiennent des informations à caractère personnel sur les passagers telles que l'identité du passager et des personnes qui l'accompagnent, les moyens de paiement utilisés, les préférences alimentaires et éventuellement la location d'un véhicule ou les réservations d'hôtel à l'arrivée. L'objectif de cette demande était de soumettre ces informations au système *CAPS II*, un système expérimental d'analyse des données, abandonné depuis, et qui avait pour but la détection automatique des passagers suspects. On peut noter que cette demande a soulevé des oppositions dans la classe politique européenne, comme en témoigne la proposition de résolution de la *Commission des libertés et des droits des citoyens, de la justice et des affaires intérieures* du parlement européen [1].

la carte, sans restriction. On peut noter que les clés valides de passagers pourraient être utilisées par le FBI pour encoder ou inférer des informations normalement cachées. Pour éviter cela, il est nécessaire de mettre en place des mécanismes non triviaux de filtrage des données transmises par l'application [49].

La généralisation de cette infrastructure permet une prise en charge transparente de la protection du fournisseur d'un code mobile. En effet, le caractère sécurisé du matériel n'a pas d'influence directe sur le code ou sur sa structure. C'est uniquement au moment du déploiement de l'application qu'il faudra s'assurer que chacune des machines hôtes présente bien les caractéristiques adéquates.

Problèmes éthiques

Un certain nombre d'architectures matérielles pour le calcul sécurisé ont déjà été proposées. Le principe est d'empêcher la personne physiquement en possession d'un périphérique de pouvoir découvrir des informations contenues dans un programme au sein de ce périphérique ou de pouvoir en perturber l'exécution.

Différents marchés et applications existent pour ces architectures, principalement dans le domaine des cartes à puce. En général, il existe alors une dissociation entre le propriétaire du périphérique et son porteur. Ce sont, par exemple, les termes employés dans le domaine des cartes bancaires pour désigner, d'une part la banque – propriétaire de la carte – et d'autre part le client utilisateur d'une carte – le porteur de la carte. Dans ce cas, on peut considérer que la banque fournit un périphérique à son client qui peut l'utiliser pour effectuer des paiements. Dans le même temps, la banque se protège de l'utilisation frauduleuse, par le porteur, de son bien. Plus collectivement, c'est l'ensemble du système de paiement par cartes bancaires qui est protégé.

Dans ce cas, le porteur en possession du périphérique n'en a pas la propriété et ce qui lui est facturé correspond au service dont il bénéficie, service dont la protection lui est utile et dont les fonctionnalités et la portée sont limitées et clairement définies. Dans le cas d'une approche généraliste comme celle que nous proposons pour le calcul sécurisé sur une grille de cartes à puces, c'est du **propriétaire** du matériel que le code est protégé. Le propriétaire a été facturé pour un matériel mais ce matériel ne lui fait pas confiance et lui interdit l'accès au code qu'il exécute. Les applications que nous avons mises en œuvre pour illustrer notre plate-forme ne soulèvent pas vraiment de problèmes éthiques, le fait que le calcul de Mandelbrot soit protégé du propriétaire du matériel sur lequel il s'exécute n'est pas, en soit, très problématique. Mais les applications peuvent aller beaucoup plus loin dans l'exploitation de ce mécanisme et cela peut devenir gênant.

Le processeur TrustNo 1 [77], par exemple, poursuit un objectif similaire : protéger le code qu'il exécute contre le propriétaire de la machine. Dans le cas de TrustNo 1, l'objectif clairement poursuivi est la protection des logiciels contre le piratage. Des fonctionnalités de révocation de licence sont par exemple proposées pour empêcher l'exécution d'un programme sur un processeur. L'ampleur des possibilités offertes aux logiciels peut

devenir inquiétante. Cette inquiétude s'est notamment manifestée dans les nombreuses polémiques autour de la spécification TCPA – Trusted Computing Platform Alliance. L'objectif de TCPA est de définir un périphérique sécurisé intégré aux ordinateurs de type PC permettant de stocker, entre autres, les clés privées d'un utilisateur et de les utiliser uniquement en interne. C'est-à-dire que toutes les fonctions de cryptographie asymétrique sont déléguées à ce périphérique, qui est le seul à connaître les clés privées qu'il stocke. L'intérêt est alors d'empêcher toute application s'exécutant sur l'ordinateur d'accéder à ces clés privées, en particulier les applications dangereuses de type virus. A la différence de l'architecture que nous avons mise en place, cette puce ne permet aucun contrôle sur l'exécution des programmes qui continuent de se dérouler sur un processeur classique, non sécurisé. De plus, elle n'est pas résistante aux attaques physiques : TCPA a pour but la protection du propriétaire du matériel et pas celle du code exécuté. Malgré cela, une polémique est née du fait qu'elle puisse être déployée massivement et éventuellement détournée de ses objectifs. On pourra lire dans [109] une défense de la plate-forme TCPA et, dans les références qui y sont citées, une critique de cette plate-forme et des dangers qu'elle peut représenter.

Lorsqu'on se propose de protéger un code, on doit se poser des questions d'ordre éthique. Quel est l'objectif poursuivi par cette protection ? Dans le cas des cartes bancaires, la protection du périphérique protège son propriétaire – la banque – du porteur. D'autre part, cette protection bénéficie au porteur, en particulier en cas de vol, et la nécessité de sécuriser cette application est communément admise. Dans le cas d'un calcul sur Internet auquel des utilisateurs décident volontairement de participer, la protection du code vis-à-vis des propriétaires du matériel sur lequel il est exécuté présente, là aussi, un intérêt qui peut être collectivement accepté : si un utilisateur – par définition honnête – participe à un projet par charité, il ne souhaite probablement pas que les résultats de ce projet soient compromis par des participants malhonnêtes. De même, s'il est rémunéré pour sa participation sincère et honnête, il ne veut pas que d'autres soient payés alors qu'ils fraudent. Dans un cas comme dans l'autre, il semble réaliste de penser que les utilisateurs puissent accepter que leur machine ne leur fasse pas confiance dans la mesure où c'est la contrepartie nécessaire pour garantir que les autres ne pourront pas frauder.

Tant que l'utilisateur voit un intérêt à ce qu'un code soit protégé des gens en qui il n'a pas confiance et, par réciprocité, de lui-même, cette protection ne pose pas de problème. Mais les dérives sont faciles : on peut imaginer un programme de comptabilité qui stocke les informations de l'utilisateur de façon cryptée. S'il est impossible de retrouver, au moins par ingénierie inverse, l'algorithme et les clés de cryptage utilisées, cela contraint l'utilisateur à se servir de cette application pour consulter sa comptabilité. Celle-ci peut alors prendre le contrôle de l'utilisateur et de ses données de façon totale ou partielle. Dans le même esprit, il devient possible d'implanter une règle ou une simple décision au sein même d'un logiciel et d'imposer ainsi son respect systématique et aveugle.

Le calcul sécurisé sur la grille nécessite l'exploitation d'une architecture, comme celle à base de cartes à puce développée dans notre équipe, protégeant le code vis-à-vis du propriétaire du matériel sur lequel il s'exécute. Dans le même temps, il est nécessaire que

le matériel empêche une application de prendre le contrôle des données de l'utilisateur. La frontière entre ces deux situations est extrêmement fine. Il faut donc rechercher un compromis permettant de fournir une infrastructure de calcul sécurisé sur la grille sans pour autant nuire à l'indépendance de l'utilisateur et à son autorité sur sa machine et sur ses données.

1.3.3 Conclusion : prise en charge transparente de la confiance

La coopération entre systèmes développés et administrés par des organisations distinctes et éventuellement concurrentes induit des problèmes de sécurité. La question de la sécurité des systèmes d'information est un domaine vaste que nous n'avons pas traité ici. Nous nous sommes plutôt focalisés sur la sécurité des codes mobiles. En effet, ceux-ci soulèvent des enjeux intéressants et spécifiques en terme de sécurité et de confiance. La gestion transparente de cette confiance semble réaliste dans la mesure où ces problèmes, soit se retrouvent dans la programmation locale, soit sont orthogonaux à l'exécution des programmes.

En ce qui concerne la protection du fournisseur du code mobile, les approches logicielles nécessitent une écriture spécifique du code pour se protéger ou détecter des corruptions dues à l'entité d'exécution. En revanche, dans le cas d'une approche matérielle comme celle qui a été mise en place par notre équipe, la protection n'a pas d'influence directe sur le code ou sur sa structure. C'est uniquement au moment du déploiement de l'application qu'il faudra s'assurer que chacune des machines hôtes présente bien les caractéristiques adéquates. Cette solution nécessiterait un déploiement, à grande échelle, d'architectures matérielles sécurisées mais nous avons vu les problèmes éthiques que cela soulève.

Pour la sécurité de l'entité d'exécution, les choses ne peuvent être totalement transparentes. En particulier, le code mobile doit être prévenu lorsqu'une opération qu'il souhaite effectuer lui est refusée. Cependant, ce problème n'est pas spécifique aux systèmes distribués mais existe déjà dans le cas local. Si le système de protection utilisé correspond à un système déjà présent dans le cadre de la programmation locale, comme c'est le cas la plupart du temps, le problème de la transparence de ce mécanisme ne se pose finalement pas.

1.4 Prise en charge de la mémoire répartie

Les différents éléments d'un système distribué, placés sur des machines distinctes, ont chacun leur propre espace mémoire. Il faut des outils pour permettre l'échange de données entre ces espaces mémoire séparés. Nous allons en traiter trois : l'échange de messages, l'appel de procédure à distance et la mémoire virtuellement partagée. Nous reviendrons plus en détail sur les messages actifs et l'appel de routine à distance au chapitre 2, dans la mesure où ils sont directement en relation avec la bibliothèque JToe que nous présenterons au chapitre 3.

1.4.1 Echange de messages

L'échange de messages est vraisemblablement la méthode la plus bas niveau, ou en tout cas la plus proche du modèle d'exécution sous-jacent. Pour qu'une machine A connaisse une information contenue sur une machine B, cette dernière doit envoyer un message contenant cette information à la machine A. Cet envoi de message est explicite au niveau du paradigme de programmation et doit être pris en charge par le développeur. Il existe de nombreuses bibliothèques et de nombreux outils pour l'envoi de messages. Parmi ceux là, on peut citer les *sockets* BSD, les bibliothèques de communication de groupes pour la programmation parallèle comme MPI ou PVM, ou encore les messages actifs.

Les *sockets* BSD sont probablement l'outil le plus utilisé pour développer par envoi de messages. Une *socket* représente l'extrémité d'un canal de communication. Des données peuvent y être écrites et cela aura pour effet de les transmettre à travers le réseau. Des données peuvent également être lues depuis une *socket*, elles proviennent alors d'un autre processus qui les a transmises. Il est possible d'exploiter des *sockets* en mode connecté ou en mode non connecté. Généralement, le mode connecté fonctionnera au dessus du protocole TCP et le non connecté au dessus d'UDP.

Les deux principales bibliothèques offrant, entre autres, des communications de groupe pour le calcul parallèle sont PVM et MPI. Ces deux plates-formes présentent certaines différences conceptuelles liées à leurs origines et à leurs objectifs. Cependant, en plus de routines de communication classiques point à point ou de *broadcast*, elles proposent toutes deux des primitives de communication collective basées sur les mêmes principes :

- *reduction* qui permet à un seul processeur de récupérer un ensemble de données réparties sur les différents membres du groupe pour les combiner en une seule ;
- *gather* / *scatter* qui permettent, respectivement, de rassembler sur une seule machine un ensemble de données éparpillées et inversement, d'éparpiller des données contenues sur une seule machine vers un ensemble de machines.

Enfin, le principe des messages actifs est d'indiquer, dans l'en-tête d'un message, l'adresse d'une séquence d'instructions qui sera exécutée à la réception du message. A la différence des appels de routine à distance que nous allons voir par la suite, l'exécution de cette séquence d'instructions n'a pas pour but d'exécuter un ensemble d'opérations complexes puis de transmettre un résultat mais simplement d'intégrer, le plus rapidement possible, les données contenues dans le message au sein du programme en cours d'exécution. Les bibliothèques de messages actifs sont généralement considérées comme étant de bas niveau et permettent l'implantation efficace de bibliothèques de plus haut niveau, comme MPI.

L'envoi de messages n'existe pas dans un paradigme de programmation locale, la question de sa transparence n'a donc pas de sens. Cependant, il constitue l'élément de base pour les communications dans un système distribué. C'est sur ce mécanisme que repose l'implantation des deux autres outils de gestion de la mémoire répartie que nous présentons dans la suite : la mémoire virtuellement partagée et l'appel de routine à distance.

1.4.2 Appel de *routine* à distance

La communication entre machines peut s'effectuer par l'appel de routine à distance. Dans ce cas, une machine A demande à une machine B d'effectuer un traitement et de lui retourner un résultat. En général, l'appel de routine à distance se fait grâce à des mécanismes déjà présents dans le paradigme de programmation. Nous utilisons ici le terme générique de *routine* pour désigner des procédures ou des méthodes selon le paradigme de programmation utilisé.

L'un des protocoles le plus répandu pour l'appel de procédure à distance est ONC RPC [119] – Open Network Computing Remote Procedure Call –, aussi connu sous le nom de *Sun RPC*. L'utilisation depuis le langage C de ONC RPC permet d'abstraire les communications réseau derrière une syntaxe habituelle d'appel de procédure. Cela n'en fait cependant pas un système transparent. Par exemple, lors d'un appel de procédure à distance, il est nécessaire de transmettre un argument qui identifie le service RPC auquel on s'adresse.

L'appel de méthode à distance permet d'exploiter un objet distant et d'invoquer une méthode sur cet objet. Pour cela, la notion de référence sur un objet est étendue au cadre distant, c'est-à-dire qu'une référence peut représenter un objet situé en réalité sur une machine distante. L'appel d'une méthode sur cette référence provoquera l'appel de la méthode en question à distance. Parmi les plates-formes permettant l'appel de méthode à distance, on peut citer CORBA [62] ou RMI [145].

Dans tous les cas, l'appel de routine à distance offre une abstraction au niveau du langage mais n'est pas transparent dans la mesure où la gestion de la mémoire elle-même n'est pas assurée. En particulier, les arguments sont toujours passés par recopie sur la machine cible et les résultats des appels sont également recopiés vers la machine à l'origine de l'appel, même dans le cas où la syntaxe correspondrait à un passage par référence. Pour autant, cela n'est pas masqué au développeur puisque certains éléments de programmation indiquent clairement l'aspect distant d'un appel. Il existe des plates-formes qui se proposent de le lui masquer syntaxiquement. Nous étudierons un exemple d'une telle plate-forme au chapitre 2 et nous verrons les problèmes que cela peut entraîner.

D'autres plates-formes se proposent également de masquer totalement, sur le plan syntaxique, l'aspect distant de certains appels mais prennent également en charge la gestion de la mémoire. Dans ce cas, l'appel est syntaxiquement masqué : un appel de méthode classique aura pour effet d'envoyer un message sur la machine hébergeant l'objet cible de cet appel. De plus, l'ensemble des conséquences de ces communications sera pris en charge par la plate-forme. En particulier, les arguments ne seront plus passés par copie mais bien par référence, conformément à la sémantique locale. Les champs seront accessibles et modifiables à distance, etc. De nombreux travaux ont été proposés dans le cadre de la plate-forme Java [6, 7, 45, 147, . . .].

Le fait d'effectuer ces appels distants de façon transparente nécessite de prendre en charge certains éléments liés aux processus légers. Par exemple, les appels de méthode à

distance sont synchrones mais impliquent l'existence de deux processus : le premier exécute la méthode appelante sur la machine de l'appelant, le second se trouve sur la machine de l'appelé et exécute la méthode appelée. Pourtant, pour l'application locale d'origine, les deux méthodes sont exécutées par le même processus léger. Comme nous l'avons déjà expliqué section 1.2.3 page 16, afin que les opérations sur les processus légers effectuées par l'application d'origine puissent continuer à s'effectuer dans un cadre distribué, des processus virtuels sont introduits par la plate-forme. Ces processus virtuels permettent de faire apparaître les deux processus, appelant et appelé, comme l'unique processus légers correspondant dans l'application d'origine. La pile de ces processus légers virtuels est distribuée sur plusieurs machines, en fonction de la localisation des objets dont ils ont invoqué des méthodes. Par ailleurs, l'exclusion mutuelle doit continuer à fonctionner dans ce contexte distribué. Les travaux dans ce domaine implantent donc une version distribuée des méthodes de synchronisation présentes en Java, généralement par un algorithme centralisé (Cf. section 1.2.1 page 14).

1.4.3 Mémoire virtuellement partagée

Une mémoire virtuellement partagée permet de faire apparaître un ensemble d'espaces mémoire distincts situés sur des machines distinctes d'un réseau comme une seule et même zone mémoire. Lors de la lecture d'une donnée en mémoire par un processus – donnée que nous appellerons par la suite *variable* –, le mécanisme de mémoire virtuellement partagée assure l'accès à cette variable même si elle se situe dans la mémoire d'une machine distante. Lors d'un accès en écriture, le mécanisme assure que la variable sera bien modifiée, quel que soit son emplacement.

Une mémoire virtuellement partagée implante, de façon distribuée, un modèle mémoire. Un modèle mémoire définit les règles de cohérence qui doivent s'appliquer aux variables contenues en mémoire lorsqu'elles sont lues et écrites par différents processus. Le modèle de cohérence séquentielle, par exemple, assure que toute lecture d'une variable obtiendra la dernière valeur écrite dans cette variable. Ce modèle semble plutôt naturel et logique. Il n'est pourtant pas souvent utilisé en pratique, même dans les environnements d'exécution locaux. En effet, les optimisations les plus courantes réalisées par les compilateurs, comme le réordonnancement d'instructions ou l'exploitation intensive des registres du processeur, sont en contradiction avec ce modèle. Dans le cas des mémoires virtuellement partagées, les problèmes sont encore plus importants, surtout en ce qui concerne la latence qui découle du nombre important de communications.

D'autres modèles de cohérence ont donc été proposés. Ils imposent souvent au programmeur l'utilisation de variables de synchronisation – nous les appellerons *verrous* dans la suite – et assurent la cohérence des valeurs des variables lors de l'utilisation de ces verrous. L'idée qui sous-tend cette approche est que les instructions de deux processus indépendants peuvent être ordonnancées et s'entre mêler de façon aléatoire. Sans verrou, l'ordre d'accès aux variables n'est pas déterministe. Ces modèles relâchés reposent sur

l'acquisition et la libération de verrous pour assurer la cohérence de la mémoire. Schématiquement, chaque processus possède son propre espace mémoire privé et travaille uniquement dans cet espace. Il met à jour son espace privé lorsqu'il acquiert un verrou et il *publie* les modifications réalisées dans sa mémoire privée, c'est-à-dire que les autres processus pourront avoir accès à ces valeurs, lorsqu'il libère un verrou.

Parmi ces modèles, on peut citer le modèle à cohérence faible – *weak consistency* [125, p. 308] – qui ne différencie pas l'acquisition ou la libération d'un verrou. Dans ce cas, l'accès à une variable de synchronisation correspond aux deux opérations simultanément. Le modèle *entry consistency* [125, p. 313] introduit cette distinction mais une variable partagée entre processus doit être explicitement déclarée comme telle par le programme et associée à un verrou. Lorsqu'un processus acquiert un verrou, il a la garantie de lire la dernière valeur *publiée* pour la variable associée. Cela nécessite d'associer un verrou à chaque variable partagée et d'acquérir le verrou de chacune des variables avant d'y accéder. Le modèle *lazy release consistency* [75] fonctionne sur le même principe mais aucune variable n'est associée à un verrou, c'est l'ensemble des variables modifiées qui est synchronisé. Cela simplifie la programmation par rapport au modèle *entry consistency* mais nécessite que l'implantation identifie les variables modifiées par le processus.

L'implantation d'une mémoire virtuellement partagée transparente implique le respect d'un modèle de cohérence mémoire existant dans un paradigme de programmation locale. Comme nous l'avons vu plus haut, les paradigmes de programmation locaux intégrant la notion de processus léger ne respectent pas la cohérence séquentielle. C'est le cas, par exemple, du langage Java qui permet donc d'envisager une implantation distribuée transparente de son modèle mémoire.

La plate-forme Java intègre la notion de processus léger. Elle doit donc spécifier les règles qui régissent les interactions entre ces derniers et la visibilité des opérations qu'ils effectuent en mémoire. Ces règles sont définies par le modèle mémoire de la plate-forme Java et elles ont pour objectif de garantir un comportement cohérent des applications sur toutes les architectures. Le modèle mémoire Java a d'abord été décrit dans [54] puis dans le JSR 133 [85] utilisé dans la plate-forme Java depuis la version 1.5. Nous donnons maintenant une description intuitive de ce modèle mémoire, pour une description plus précise et plus complète, le lecteur est renvoyé au JSR 133 [85].

Dans ce modèle mémoire les variables sont stockées dans une mémoire centrale. Ces variables sont les champs des objets, les tableaux et plus généralement, l'ensemble des données stockées dans le tas. Chaque processus léger possède son propre espace mémoire de travail privé qui contient des copies des variables sur lesquelles il travaille. Sur un plan pratique, cet espace mémoire privé peut être comparé aux registres du processeur, au cache processeur sur une machine multi-processeurs ou, pourquoi pas, à la mémoire de la machine locale dans le cas d'une mémoire répartie virtuellement partagée. Un certain nombre de règles sont définies par le modèle mémoire afin de synchroniser le contenu de l'espace privé d'un processus léger avec la mémoire centrale. C'est en particulier le cas lors de l'acquisition ou de la libération d'un verrou. Lors de l'acquisition d'un verrou, il devra mettre à jour son espace privé avec les valeurs contenues dans la mémoire centrale.

Dans le cas de la libération d'un verrou, il devra recopier les données modifiées de son espace privé dans la mémoire centrale. D'autres contraintes sont imposées, par exemple la lecture ou l'écriture de variables déclarées *volatile* provoque également une synchronisation avec la mémoire globale. De même les champs déclarés *final* reçoivent un traitement particulier.

Cela signifie que l'accès à des données partagées entre plusieurs processus légers, en l'absence de synchronisation entre ceux-ci, peut aboutir à des comportements surprenants comme dans cet exemple extrait du JSR 133 dans lequel, initialement, A et B valent 0 :

Processus 1	Processus 2
`r2 = A;`	`r1 = B;`
`B = 1;`	`A = 2;`

A la fin de l'exécution, on peut avoir `r2 == 2` et `r1 == 1`. En effet, en l'absence de synchronisation explicite, le compilateur va optimiser processus par processus. Pour chacun d'entre eux, les instructions sont indépendantes et le compilateur peut donc décider de les réordonner.

En observant ce modèle mémoire, on constate qu'il est très proche de modèles mémoire souples destinés aux mémoires réparties virtuellement partagées, comme le modèle *Lazy Release Consistency*[75], par exemple. Mihai Surdeanu et Dan Moldovan ont prouvé que leur modèle mémoire OMW, basé sur le modèle *Lazy Release Consistency* était équivalent, dans le cas des programmes correctement synchronisés, au modèle mémoire Java originel [124]. Etant données les évolutions qui ont été proposées dans le JSR 133, il semblerait que cette équivalence soit maintenant valable même dans le cas de programmes non correctement synchronisés.

Jackal [133], Hyperion [5] et DISK [124] proposent tous trois une implantation distribuée du modèle mémoire Java. Jackal et Hyperion compilent le code Java en code natif. Dans le cas de Jackal, des instructions sont insérées dans le code durant cette compilation pour implanter le mécanisme de cohérence mémoire. En particulier, des tests sont insérés pour vérifier la disponibilité des données avant les accès mémoire. Un certain nombre d'optimisations sont mises en œuvre pour diminuer le nombre de ces tests. Dans le cas d'Hyperion, le code généré exploite la plate-forme générique de mémoire virtuellement partagée DSM-PM^2 [4]. Différents modèles mémoires peuvent être implantés au dessus de DSM-PM^2 et une implantation du modèle mémoire Java est proposée. Enfin, DISK [124] modifie la machine virtuelle Kaffe. Ces trois approches reposent toutes sur un protocole de cohérence mémoire de type *lazy release consistency*.

1.4.4 Conclusion : prise en charge transparente de la mémoire répartie

La gestion de la mémoire répartie peut se faire via l'échange explicite de messages entre les différentes machines, auquel cas les choses ne sont pas transparentes. Elles peuvent également se faire grâce à l'appel de routine à distance, ce qui offre une certaine abstraction. Cependant, nous verrons au chapitre 2 que cette abstraction syntaxique ne doit pas aller trop loin. Nous proposerons au chapitre 3 une abstraction des communications sous la forme de transfert d'objets. Cependant, dans ce cas, la répartition de la mémoire restera explicite.

La véritable méthode pour gérer de façon transparente la mémoire répartie consiste non seulement à masquer cette répartition sur le plan syntaxique mais également à prendre en charge les conséquences sémantiques de cette répartition. Dans le domaine des appels de routine à distance, nous avons vu que certaines plates-formes proposaient un masquage total des communications engendrées par les appels de routine à distance.

Les mémoires virtuellement partagées offrent également une certaine transparence vis-à-vis de la gestion de la mémoire. Cependant, nous avons vu que, pour les limiter les communications et la latence induite, elles mettent généralement en œuvre des modèles mémoire relâchés. Une mémoire virtuellement partagée ne peut être transparente qu'à la condition que le modèle mémoire qu'elle implante corresponde à un modèle mémoire local. C'est le cas du modèle mémoire *lazy release consistency* qui est compatible avec le modèle mémoire du langage Java [85], par exemple. Des mémoires virtuellement partagées totalement transparentes pour le langage Java ont donc été proposées [5, 124, 133].

La présentation des mémoires virtuellement partagées nous a permis d'entrevoir certains mécanismes de gestion de la latence. Nous revenons sur cette caractéristique dans la prochaine section.

1.5 Prise en charge de la latence

La latence correspond à un phénomène physique. Dans le cas des systèmes distribués, il s'agit du temps nécessaire pour qu'une unité d'information transite d'une machine A vers une machine B.[7] Comme nous venons de le voir, la prise en charge de la latence est liée à celle de la mémoire répartie : une machine B a besoin d'une donnée disponible sur une machine A, cela nécessite une communication qui est soumise à la latence. L'objectif de la prise en charge de la latence est d'en limiter l'impact soit en recouvrant les communications par d'autres opérations, soit tout simplement en limitant les communications. Nous allons présenter ces deux approches. A cette occasion nous aborderons le principe des appels asynchrones de routine à distance sur lesquels nous reviendrons en profondeur au chapitre 4.

[7] De façon générale, nous inclurons dans la latence le temps nécessaire au transfert d'une quantité de données, temps qui, en réalité ne dépend pas que de la latence mais également du débit.

1.5.1 Limitation des communications

La limitation des communications peut être réalisée grâce à l'utilisation de mémoires cache ou par la mise en œuvre de politiques de placement.

Placement

Une politique de placement s'intéresse à la localisation des processus et des données d'une application sur un ensemble de machines. On peut considérer principalement deux types de politiques, celles dont l'unité de placement est le processus et celles dont l'unité est la donnée. Les politiques dont l'unité de placement est le processus placent les processus sur les différentes machines disponibles. Celles dont l'unité est la donnée y placent... les données. Dans un cas comme dans l'autre, des mécanismes sont mis en œuvre pour s'assurer que processus et donnée sont bien disponibles sur la même machine lorsque le premier a besoin de la seconde pour effectuer une opération.

Le placement de données se retrouve principalement sur les plates-formes d'appel de routine à distance. Dans ce cas, la donnée placée est un service ou un objet. L'objectif du placement sera de limiter le nombre d'appels distants en rassemblant les composants – objets ou services – communicant beaucoup entre eux [24]. Le placement peut s'effectuer soit au moment de la création du composant, soit par la suite, via un mécanisme de migration. Dans le cas des objets, le placement peut se baser sur la détection d'un modèle de conception : pour une *fabrique*, les objets *fabriqués* seront construits directement sur la machine de l'appelant plutôt que sur la machine hébergeant la fabrique [7] ; la migration d'objets permettra d'anticiper l'envoi d'objets dans des modèles de type *producteur/consommateur* [45].

Le placement de processus poursuit le même objectif de diminuer les communications entre processus. Ainsi, deux processus communicant beaucoup seront rapprochés l'un de l'autre. L'autre objectif complémentaire est l'utilisation de l'ensemble des ressources de calcul disponibles. Bien que celui-ci puisse également être poursuivi par une politique de placement de données, les choses sont rendues plus claires ici, dans la mesure où l'unité de placement – le processus – correspond à l'unité de calcul – le processeur.

Quelle que soit l'unité de placement – processus ou donnée – considérée par une politique, une instruction ne peut s'exécuter que si, à la fois, la donnée qu'elle utilise et le processus qui l'exécute sont présents sur la même machine, au même instant. Par exemple, dans le cas d'une politique dont l'unité de placement est le processus, les différents processus de l'application seront placés sur les différentes machines disponibles mais il faudra s'assurer que les données qu'ils utilisent soient disponibles sur leur machine respective au moment où ils en auront besoin. Pour cela, l'utilisation d'une mémoire cache peut être envisagée.

Mémoires cache

L'utilisation de mémoires cache permet de réduire la latence. Cette méthode est depuis longtemps utilisée dans les processeurs. Elle est également utilisée dans les mécanismes de mémoire virtuellement partagée que nous avons vus section 1.4.3 page 32.

Dans le cas des processeurs, l'écart entre le temps d'exécution d'une instruction par le processeur et le temps d'accès à la mémoire est tel que le processeur est obligé d'*attendre* au cours de ces accès. Pour éviter cela, des mémoires cache de taille réduite mais à accès très rapide sont utilisées. Les lectures et les écritures s'effectuent directement dans cette mémoire cache. Ainsi, l'écriture d'une donnée ne subit plus la latence liée à l'accès à la mémoire. Pour la lecture, en cas de défaut de cache, c'est-à-dire si la donnée n'est pas présente dans la mémoire cache, le processeur devra alors effectivement accéder à la mémoire et attendra le temps nécessaire à la réalisation de cet accès. Au passage, la donnée sera stockée dans le cache et les prochains accès en lecture seront directement effectués depuis celui-ci.

A l'échelle d'un système distribué, on peut voir la mémoire locale d'une machine comme de la mémoire cache. Ainsi, les accès en lecture et en écriture à une donnée peuvent s'effectuer directement dans la mémoire locale – indépendamment du mécanisme de cache de la mémoire mis en œuvre par le processeur que nous venons de voir. Tout comme pour les processeurs, en cas de défaut de cache, c'est-à-dire si la donnée n'est pas présente lors d'une lecture, celle-ci devra être récupérée à distance, ce qui entraîne une latence liée au réseau. Ce principe d'utilisation de la mémoire locale comme un cache dans un système distribué sera plus ou moins transparent, selon qu'il sera implanté par une mémoire virtuellement partagée ou explicitement, au sein du programme, par l'envoi de messages.

Dans le cas du cache processeur sur une seule machine mono-processeur, la gestion du cache correspond à peu près à ce que nous venons de présenter. Mais dès que les données sont partagées par plusieurs processeurs – machines multi-processeurs – ou par plusieurs machines – système distribué –, chacun possédant sa propre mémoire cache, il est indispensable de mettre en œuvre des mécanismes assurant la cohérence des données entre les différents caches. Schématiquement, lorsqu'un processeur modifie une donnée dans son cache, les autres devront, par la suite, lire cette nouvelle valeur plutôt que celle qui était stockée dans leur propre cache.

Si les contraintes de cohérence imposées entre les différents caches sont trop fortes alors la mémoire cache ne présente que peu d'intérêt puisque chaque modification va engendrer des communications pour mettre à jour les autres caches. Il est donc nécessaire de définir des modèles de cohérence relâchée. Nous avons présenté certains modèles de cohérence relâchée pour les mémoires virtuellement partagées à la section 1.4.3 page 32. On retrouve le même type d'approche dans le cas des machines multi-processeurs. Le modèle mémoire Java également présenté section 1.4.3 page 32 a d'ailleurs pour objectif de s'adapter à ces modèles mémoire relâchés. D'autres types de cohérence relâchée sont mis en œuvre dans les systèmes distribués et peuvent parfois être directement intégrés à la

logique du programme. C'est le cas, par exemple du modèle de synchronisation des tables entre serveurs primaires et serveurs secondaires dans le cas du DNS (Cf. 1.1.2 page 11). Le protocole HTTP/1.1 [48] définit également les règles permettant la mise en mémoire cache de pages web.

La gestion de la latence par un système de cache n'est généralement pas transparente dès qu'au moins deux caches doivent être maintenus cohérents. Par exemple dans le cas d'une mémoire virtuellement partagée, cela nécessite la définition d'un modèle de cohérence mémoire relâché que le programmeur doit respecter. Mais on doit également remarquer que le même type de modèles relâchés a dû être proposé dans les machines multi-processeurs. Dans le cas des programmes séquentiels, cela n'a pas d'influence mais les programmes locaux utilisant des processus légers doivent intégrer, dans leur conception, ces modèles mémoire relâchés.

Lorsque la prise en charge de la latence par des mécanismes de mémoire cache existe déjà dans les paradigmes de programmation locale, l'implantation en distribué de ces mécanismes permet d'assurer la prise en charge des applications conçues selon un paradigme de programmation locale dans un modèle d'exécution distribuée. Nous en avons vu des exemples pour le langage Java à la section 1.4.3 page 32.

1.5.2 Recouvrement des communications

Le recouvrement des communications permet de diminuer l'impact de la latence. Pendant que des données sont reçues ou transmises, d'autres instructions continuent d'être exécutées sur la machine. Nous allons voir trois méthodes de recouvrement des communications : (1) le *préchargement* qui consiste à demander le chargement d'une donnée avant qu'on en ait réellement besoin. On recouvre les communications engendrées par le chargement de cette donnée par d'autres opérations ; (2) l'utilisation des *processus légers* qui consiste à avoir plusieurs processus légers prêts à s'exécuter sur une machine donnée. Lorsqu'un de ces processus légers effectue une communication, un autre processus utilise le processeur et recouvre les communications du premier ; (3) l'*appel asynchrone de routine à distance* qui permet de continuer l'exécution du processus en cours pendant qu'une routine est exécutée sur une autre machine.

Préchargement

Le préchargement permet le recouvrement des communications en demandant une donnée avant d'en avoir besoin. Il s'utilise généralement avec une mémoire cache. Lors d'un accès en lecture à une donnée, si celle-ci n'est pas présente en cache, il est nécessaire d'*attendre* son chargement. Pour éviter cela, elle peut être préchargée en prévision de son utilisation. En attendant, les instructions indépendantes de cette donnée peuvent être exécutées normalement. Si tout se passe bien, la donnée est déjà en cache au moment où on en a besoin.

On distingue généralement deux types de préchargement [66]. Le préchargement liant – *binding prefetching* – et le préchargement non liant – *non-binding prefetching*. Dans le cas du préchargement liant, une fois une donnée préchargée, celle-ci ne peut plus être modifiée par un autre processeur. Plus précisément, toute modification faite entre le moment du préchargement et le moment effectif d'utilisation sera invisible et donc synonyme, selon le modèle de cohérence, de corruption de donnée. Cela signifie qu'une donnée ne peut être préchargée qu'à partir du moment où sa valeur ne sera plus modifiée avant son utilisation. A l'inverse, le préchargement non liant autorise la modification d'une donnée entre le moment de son préchargement et celui de son utilisation. Cette modification devra alors être prise en compte juste avant l'utilisation de la donnée.

On retrouve des méthodes de préchargement dans la plupart des systèmes à base de cache. Le logiciel *Squid* [118], par exemple, qui met en place un cache de pages web, dispose d'un module exploitant le préchargement : lorsqu'on accède à une page via *Squid*, les pages référencées par celle-ci sont préchargées. De même, la norme *HTML 4.01*, proposée par le W3C, spécifie l'élément `LINK` qui peut être de type `next` [139]. Un tel élément spécifie la page suivante à consulter dans un parcours raisonnable du site. Le navigateur *Mozilla* exploite ce marqueur et l'étend en ajoutant le type explicite `prefetch` [102]. Pendant que l'utilisateur consulte une page internet, le navigateur précharge les liens identifiés par ces marqueurs. Il recouvre ainsi les communications par la lecture.

Les processeurs ont également recours au préchargement. Le processeur *Pentium III*, par exemple, propose l'instruction assembleur `prefetcht0` qui a pour effet de déclencher le préchargement d'une donnée de la mémoire vers les mémoires cache du processeur.

Le préchargement peut aussi être utilisé dans le cadre de la mémoire virtuellement partagée. Son étude dans ce contexte a fait l'objet de nombreux travaux [2, 11, 66, 95, 104]. Dans le cas du modèle de cohérence *Lazy Release Consistency*, par exemple, le préchargement permet d'obtenir l'ensemble des données auxquelles on accédera dans une section critique avant d'y entrer. Pour cela, il est possible d'avoir recours à un mécanisme de préchargement liant. Cependant, il ne pourra être déclenché qu'à l'entrée en section critique, ce qui augmentera le délai associé à l'acquisition d'un verrou. Cela peut être très pénalisant dans le cas de sections critiques courtes [11]. A l'inverse, un préchargement non liant pourra être effectué à n'importe quel moment avant l'entrée en section critique. Cependant, il sera nécessaire de remettre la donnée à jour avant son utilisation réelle si elle a été modifié entre temps [95].

Le préchargement peut être déclenché par des instructions de préchargement insérées dans le programme par un compilateur ou par le développeur, avec l'effort supplémentaire que cela suppose [95]. Il peut également se baser sur une observation du programme pendant l'exécution. Adaptive++ [104], par exemple, tente de repérer des motifs dans l'exécution d'une application en analysant les accès mémoire entre deux barrières. S'ils se répètent, alors, au début d'une nouvelle phase de calcul, Adaptive++ va précharger les données qu'il avait été nécessaire de charger durant la phase précédente. Sinon, si les zones mémoire auxquelles on accède sont réparties de façon régulière, alors Adaptive++

va les précharger en anticipant les accès futurs.

Le préchargement de données inutiles engendre des communications réseau superflues qui peuvent dégrader les performances des applications. Le préchargement peut aussi conduire à la concentration, dans un laps de temps réduit, de communications qui, au départ, étaient réparties dans le temps. Cela peut aboutir à des contentions au niveau du réseau, à une diminution de ses performances et par conséquent de celles de l'application [95]. Bien que le préchargement puisse contribuer à une amélioration nette des performances, s'il est utilisé pour masquer la latence de façon transparente et sans une connaissance spécifique de l'application cible, son utilisation peut aboutir à l'effet inverse de celui recherché.

Processus légers

Les processus légers permettent le recouvrement des communications : pendant qu'un processus léger attend qu'une communication s'effectue, un autre prend la main et exécute des instructions sur la machine. Alors que le préchargement oblige à connaître à l'avance les communications qui vont être nécessaires, l'avantage des processus légers est qu'a priori, aucune connaissance particulière ni aucune capacité de prédiction n'est nécessaire en ce qui concerne l'application cible.

Nous avons vu que la plupart des processeurs utilisent un mécanisme de mémoire cache et éventuellement de préchargement pour compenser la lenteur relative de la mémoire par rapport au processeur. Une alternative est l'intégration, au sein du processeur, de la notion de processus léger. Nous appellerons ces processeurs *processeurs multi-thread* [130]. Il existe différents types de processeurs multi-thread :

- IMT – *Interleaved Multi-Threading*. Une instruction d'un processus léger différent est traitée à chaque cycle.
- BMT – *Blocked Multi-Threading*. Un seul processus léger est exécuté séquentiellement jusqu'à ce qu'une opération nécessitant une attente du fait d'une latence du matériel sous-jacent – comme un accès mémoire – soit rencontrée, ce qui provoque un changement de contexte.
- SMT – *Simultaneous Multi-Threading*. Les instructions de plusieurs processus légers sont exécutées simultanément par un seul processeur superscalaire.

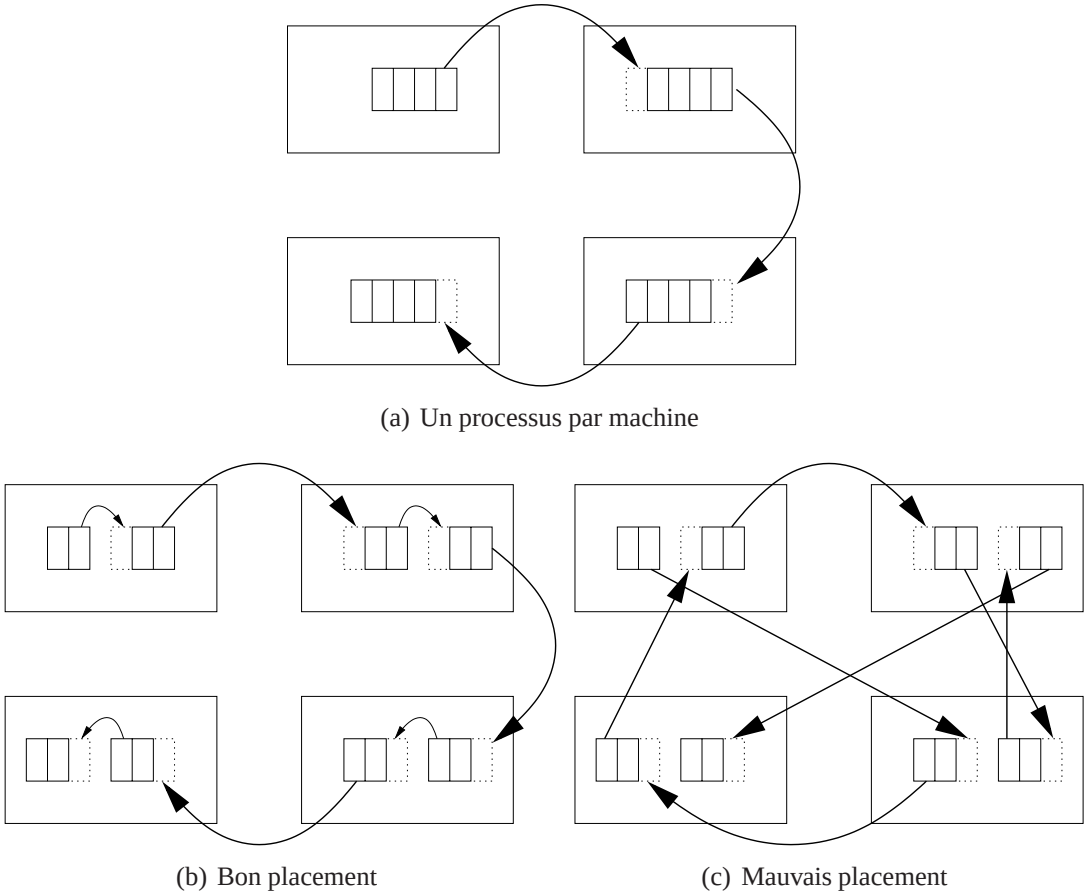
Les processeurs *multi-thread* permettent donc d'utiliser un nombre important de processus légers pour un coût de gestion, et notamment de changement de contexte, très faible. Ils permettent le recouvrement des communications par le calcul et sont d'ailleurs parfois utilisés ou recommandés pour l'implantation de mémoires virtuellement partagées [66].

Les processus légers logiciels sont gérés par le système ou par une bibliothèque. Le changement de contexte a un coût beaucoup plus important et la granularité est de ce fait moins fine que dans le cas des processeurs *multi-thread*. On pourra trouver dans [99] une étude de différentes plates-formes permettant la programmation à base de processus légers dans le cadre du calcul haute performance.

Pour qu'une application bénéficie du recouvrement des communications par le calcul grâce aux processus légers, il est nécessaire que celle-ci possède un niveau de parallélisme important. Ce parallélisme peut être algorithmique – implantation d'un algorithme parallèle avec des processus légers – ou fonctionnel – chaque processus léger possède un *rôle* dans l'application. Les paradigmes de programmation locaux intégrant la notion de processus légers, leur exploitation pour permettre le recouvrement des communications par le calcul peut donc se faire de façon transparente. Cependant, le recouvrement qui pourra être réalisé dépendra directement du degré de parallélisme de l'application. L'augmentation automatique de celui-ci n'est pas une tâche triviale.

Dans le cas de l'utilisation de processus légers pour l'implantation d'un algorithme parallèle, on peut imaginer augmenter le nombre de ces processus légers [46], par exemple en modifiant les paramètres passés au programme sur la ligne de commande, si ceux-ci servent à spécifier le nombre de processeurs disponibles [129]. Cependant, cela implique une gestion fine du placement des processus légers sur les différentes machines. Dans le cas de programmes de type SPMD dont le schéma de communication est régulier, cette méthode peut aboutir à une augmentation des communications [47] et un mécanisme de type préchargement sera, de par cette régularité des communications, mieux adapté.

Considérons, par exemple, un calcul découpé en étapes opérant sur un tableau. On suppose qu'au début de chaque étape, chaque processus léger effectue une copie de la colonne adjacente à la partie du tableau qu'il gère. Dans la logique originale du programme, on a un processus léger par processeur sur la machine locale. Le tableau est donc découpé en n zones, n étant le nombre de processeurs. A chaque étape, les n processus copient la valeur d'une colonne adjacente. Supposons qu'on souhaite maintenant exécuter ce programme sur un système distribué. Considérons la méthode de distribution simple qui place un processus léger sur chaque machine. La plate-forme se charge d'assurer les communications lorsque c'est nécessaire. Sur la figure 4(a) page ci-contre, quatre machines distinctes hébergent quatre processus. Chaque processus exécute un processus léger de l'application d'origine et possède une zone du tableau. A la fin d'une étape de calcul, la copie de la colonne adjacente – représentée en pointillés – déclenche une communication. Le transfert des colonnes adjacentes est représenté par les flèches. On considère maintenant qu'on utilise m processus légers pour exécuter ce même algorithme, avec $m = x * n$, x étant un entier supérieur à 1. Si on place des processus légers qui travaillent sur des zones de tableau contiguës sur la même machine, comme sur la figure 4(b) page suivante, pendant que l'un d'entre eux attend qu'on lui communique la zone adjacente dont il a besoin, les autres peuvent effectuer des calculs après avoir simplement copié leurs zones adjacentes présentes localement. On aura toujours n communications à chaque étape pour obtenir les colonnes adjacentes distantes. Si au contraire, on place ces processus légers de façon aléatoire comme sur la figure 4(c) page ci-contre alors on risque d'avoir, dans le pire des cas, x processus légers par nœud dont la colonne adjacente est distante. On aura donc $x * n$ communications à chaque étape au lieu de n et aucun recouvrement puisque tous les processus légers seront en attente de leur colonne. Une telle augmentation du nombre de



(a) Un processus par machine

(b) Bon placement

(c) Mauvais placement

FIG. 4 – Ajout de processus légers

processus légers sans prise en charge spécifique du placement aboutit alors à une dégradation significative des performances du programme.

Dans le cas d'une application fonctionnellement parallèle, les différents processus légers auront différents rôles. Ce type de parallélisme est parfaitement adapté au recouvrement des communications par le calcul. C'est même une des possibilités offerte par la méthode ELM [112] – *Entity Life Modelling*– que de détecter et d'effectuer les découpages qui permettront d'obtenir une application recouvrant ses communications par du calcul. Par contre, une fois les différents rôles établis, l'augmentation automatique du degré de parallélisme n'est pas évident.

Les processus légers déjà existants au sein d'une application locale peuvent être exploités pour recouvrir les communications de façon transparente. Par contre, si on souhaite augmenter leur nombre, le placement de ces derniers doit être considéré avec attention pour trouver un compromis avec l'augmentation des communications comme sur l'exemple de la figure 4.

Appel asynchrone de *routine* à distance

Nous avons vu les mécanismes d'appel de routine à distance à la section 1.4.2 page 30. Dans toutes les plate-formes que nous avons décrites et sur lesquelles nous reviendrons plus en détail au chapitre 2, les appels sont effectués de façon synchrone : l'appelant est bloqué et ne peut continuer son exécution tant que le résultat de sa commande ne lui est pas parvenu. Ce temps d'attente inclut le délai nécessaire au transit de la commande sur le réseau, à son exécution puis au transit du résultat. L'appel asynchrone de routine à distance a pour but de recouvrir ce délai.

L'asynchronisme des appels de routine à distance peut s'effectuer de différentes façons [44]. L'asynchronisme peut être réalisé côté serveur, c'est-à-dire sur la machine de l'appelé. Dans ce cas, l'appelant enverra son appel de routine de façon synchrone et attendra un accusé de réception avant de continuer son exécution. Le temps d'exécution de la routine ainsi que l'envoi du résultat pourront être recouverts. Ce type d'asynchronisme permet d'assurer à l'appelant que, lorsque l'instruction suivant immédiatement cet appel est exécutée, l'appel de routine a bien été reçu par la machine distante.

L'asynchronisme peut s'effectuer côté client, c'est-à-dire sur la machine de l'appelant. Dans ce cas, l'envoi de la commande, l'exécution de celle-ci ainsi que le retour du résultat peuvent être recouverts. Par contre, aucune garantie n'est fournie au développeur quant à l'état de son appel asynchrone lorsque l'instruction qui suit immédiatement dans le programme est exécutée. Il est donc nécessaire de fournir des primitives supplémentaires permettant de vérifier si l'appel de routine a bien été reçu et, en particulier, si les données concernant cet appel ont été transmises ou non et peuvent être réutilisées ou non.

Dans le cas où l'asynchronisme est présent à la fois côté client et côté serveur, l'objectif est alors de recouvrir totalement l'appel – asynchronisme côté client – et d'assurer une certaine politique de gestion des appels, comme pour le mécanisme d'auto-protection de la section 1.2.2 page 15 – asynchronisme côté serveur.

Nous reviendrons plus en détail sur les appels asynchrones de routine à distance au chapitre 4 et sur quelques plates-formes qui offrent cette fonctionnalité. Nous étudierons la question du masquage syntaxique de l'asynchronisme d'un appel et les problèmes que cela peut soulever, notamment en ce qui concerne la gestion des exceptions. Enfin, nous présenterons au chapitre 5 la propriété d'activabilité proposée dans le cadre de la plate-forme ProActive [65] ainsi que l'extension que nous en proposons, qui permettent d'introduire automatiquement et de façon transparente des appels de méthode asynchrones dans un programme objet.

1.5.3 Conclusion : prise en charge transparente de la latence

La latence est liée à des phénomènes physiques et représente le temps nécessaire à une information pour transiter d'une machine à une autre dans un système distribué. La particularité de la latence vis-à-vis des autres caractéristiques est que la plupart des applications écrites selon un paradigme de programmation locale pourront s'exécuter sur un

modèle d'exécution distribuée – si les autres caractéristiques sont bien prises en charge – sans que la latence n'ait d'influence sur la sémantique de l'application. Ainsi, la latence ne nécessite pas forcément d'efforts particuliers dans le cadre de l'unification des paradigmes de programmation locale et distribuée, en tout cas sur le plan de la sémantique. Nous avons vu, par exemple, que certains modèles de cohérence mémoire relâchée des mécanismes de mémoire cache sont déjà intégrés aux paradigmes de programmation locale.

Les techniques de prise en charge de la latence que nous avons présentées ne peuvent pas toutes être appliquées de façon aveugle. C'est le cas du préchargement qui, s'il est mal utilisé, peut aboutir à une augmentation importante des communications. Dans le cas des processus légers, il est possible d'exploiter ceux déjà existants dans l'application pour recouvrir les communications. Par contre, si on souhaite augmenter le nombre de processus légers, le placement de ces derniers doit être considéré avec attention pour éviter d'accroître encore les communications comme sur l'exemple de la figure 4 page 41.

L'appel asynchrone de routine à distance est une autre approche qui permet d'atténuer l'impact de la latence dans les plates-formes d'appel de méthode à distance. Le principe est d'appeler une routine à distance sans attendre son résultat. Nous reviendrons plus en détail sur cela au chapitre 4. D'un certain point de vue, la propriété d'activabilité que nous décrirons au chapitre 5 permet d'exploiter de façon transparente les appels de routine à distance puisqu'elle introduit automatiquement de tels appels dans une application séquentielle tout en conservant la sémantique originelle. Cependant, le masquage syntaxique de l'asynchronisme pose un certain nombre de problèmes, en particulier en ce qui concerne la gestion des exceptions. Nous verrons au chapitre 4 que les mécanismes de gestion d'exception pour les appels synchrones ne sont pas adaptés aux appels asynchrones.

1.6 Unification des paradigmes de programmation locale et distribuée

Nous avons défini l'*unification de la programmation locale et distribuée* de façon utopique comme la possibilité de réaliser des programmes selon un paradigme de programmation locale et qui puissent s'exécuter de façon optimale quel que soit le modèle choisi, d'exécution locale ou distribuée. Cela implique que la plate-forme d'exécution distribuée soit capable de prendre en charge, de façon transparente, les caractéristiques que nous venons de présenter. Cette unification est possible, au mot *optimale* près.

Commençons par les **pannes partielles**. Nous avons vu qu'il était possible de les gérer de façon transparente en les transformant en pannes totales. On n'obtient malheureusement pas une solution *optimale*. En effet, la probabilité qu'un programme se termine en raison d'une panne est nettement supérieure dans le cas d'un système distribué offrant cette transparence que dans le cas d'un système local, alors qu'à l'inverse, la probabilité

que toutes les machines d'un système distribué soient en panne simultanément est beaucoup plus faible. Les méthodes de gestion de pannes comme la réplication ou les points de reprise peuvent être mises en œuvre de façon transparente pour améliorer cette situation. Cependant, cette transparence implique des hypothèses pessimistes en ce qui concerne la sauvegarde des points de contrôle ou le maintien de la cohérence entre réplicas. Vogels et al. [136] constatent que des solutions ad-hoc, spécifiques à la sémantique des applications concernées sont souvent préférées. Finalement, ces deux mécanismes permettent bien la gestion transparente des pannes partielles mais limitent l'échelle à laquelle une application pourra être déployée, que ce soit en raison de l'augmentation de la probabilité de panne ou du coût de la gestion transparente de panne.

La latence a la particularité de pouvoir être totalement ignorée sans nuire, la plupart du temps, à la sémantique de l'application. Cependant, son impact au niveau des performances peut être extrêmement nuisible et nous avons étudié quelques techniques pour sa prise en charge section 1.5 page 34, soit en limitant les communications, soit en les recouvrant. Dans le cas du recouvrement, le préchargement peut réduire la latence mais son utilisation aveugle risque d'aboutir à une saturation inutile du réseau. Les processus légers existant dans une application locale peuvent être exploités. Mais le recouvrement est alors limité par le degré de parallélisme de l'application et nous avons montré sur l'exemple de la figure 4 page 41 que l'ajout transparent de processus légers, sans un placement attentif de ces derniers afin de limiter les communications, pouvait dégrader les performances d'une application. Les mémoires cache permettent également de limiter les communications à condition d'être associées à un modèle de cohérence relâché. Nous y revenons au paragraphe suivant. L'appel asynchrone de routine à distance est une autre possibilité, nous l'étudierons ainsi que son utilisation transparente aux chapitres 4 et 5.

La mémoire répartie peut être gérée de façon transparente via un mécanisme de mémoire virtuellement partagée. Ces mécanismes sont intimement liés à la latence et à la définition de modèles mémoires relâchés qui limitent son impact. Comme nous l'avons remarqué section 1.4.3 page 32, lorsque le modèle mémoire exploité par une mémoire virtuellement partagée correspond au modèle mémoire d'un paradigme de programmation locale, comme c'est le cas du modèle *lazy release consistency* et du modèle mémoire Java, l'utilisation d'une mémoire virtuellement partagée devient transparente. L'appel de routine à distance peut également permettre d'abstraire les communications. Nous avons évoqué section 1.4.2 page 30 certaines plate-formes qui exploitent cela pour proposer un masquage total des communications. D'autres se contentent d'un masquage syntaxique et la sémantique des appels reste celle d'appels distants. Nous reviendrons au chapitre 2 sur les problèmes que soulève cette dernière approche.

En ce qui concerne **la confiance**, nous nous sommes intéressé à la confiance entre le fournisseur d'un code mobile et l'entité d'exécution de ce code. La protection du fournisseur vis-à-vis de l'entité d'exécution impose d'utiliser un moyen pour s'assurer que l'entité d'exécution ne perturbe pas l'exécution du code. Cette garantie peut être fournie de façon transparente dans le cas où elle repose sur une infrastructure matérielle sécurisée comme celle que nous avons décrite section 1.3.2 page 23. C'est uniquement

au déploiement de l'application qu'il sera nécessaire de s'assurer que chacune des machines hôtes fournit ou non le matériel approprié. La seule alternative possible est de se restreindre à l'utilisation de machines gérées par des organismes dans lesquels on a totalement confiance. Le problème de la gestion transparente de la sécurité de l'entité d'exécution, quant à lui, ne se pose pas dans la mesure où les mécanismes de protection sont déjà présents dans les systèmes locaux.

Enfin, les problèmes de gestion de **la concurrence** sont déjà présents dans les systèmes locaux. La prise en charge de cette caractéristique dans un système distribué peut donc se faire de façon totalement transparente. Cependant, une application locale à processus légers peut faire des hypothèses sur le nombre de processus légers accédant simultanément à une ressource. Il est essentiel que la plate-forme d'exécution respecte ces hypothèses en limitant la concurrence sur les ressources de l'application à celle déjà présente dans l'application d'origine.

L'unification qui consiste à gérer de façon totalement transparente les caractéristiques des systèmes distribués est donc possible mais avec les limites que nous avons décrites pour chacune des caractéristiques : gestion sous-optimale des pannes partielles ; compatibilité entre le modèle mémoire local et un modèle mémoire relâché pour mémoire virtuellement partagée ; confiance dans les machines exploitées pendant l'exécution ; limitation des accès aux ressources de l'application.

Une autre voix pour l'unification des paradigmes de programmation locale et distribuée consiste à enrichir le paradigme de programmation locale avec des outils adaptés aux caractéristiques des systèmes distribués. Cette solution doit, a priori, être rejetée puisque le résultat produit serait un paradigme de programmation distribuée et pas un paradigme unifié. Pourtant, on constate une convergence des deux paradigmes au fur et à mesure de leur évolution. Sans que cela ait forcément été motivé par l'unification, un certain nombre d'outils ont été ajoutés aux paradigmes de programmation locaux : processus légers, exclusion mutuelle, modèles mémoire relâchés, gestion des droits, etc.

Une caractéristique pour laquelle cette convergence n'a pas encore été observée est celle des pannes partielles. Rien aujourd'hui, dans un paradigme de programmation locale, ne semble permettre la prise en charge de quelque chose qui puisse se rapprocher, de près ou de loin d'une panne partielle. L'arrêt brutal d'un processus léger dans une application locale peut laisser cette dernière dans un état incohérent, en particulier si cet arrêt se produit en plein milieu d'une section critique. La gestion de l'arrêt brutal d'un processus léger est un problème ouvert et les seules solutions actuelles consistent à empêcher ce type d'arrêt [123, 128]. Une solution à ce problème sera peut-être source de convergence sur la question des pannes partielles.

Première partie

Prise en charge de la mémoire répartie

Chapitre 2

Mémoire répartie : quelques approches pour sa gestion explicite

Les différents éléments d'un système distribué, placés sur des machines distinctes, ont chacun leur propre espace mémoire. Il faut des outils pour permettre l'échange de données entre ces espaces mémoire séparés. Nous avons déjà évoqué en 1.4 page 28 les trois mécanismes que sont : l'échange de messages, l'appel de routine à distance et la mémoire virtuellement partagée. Au chapitre suivant, nous présenterons l'API JToe, laquelle propose une prise en charge explicite de la mémoire répartie sous la forme de transfert d'objets. Nous revenons auparavant sur les mécanismes de gestion explicite de la mémoire partagée existants qui nous semblent se rapprocher le plus de JToe.

Nous présenterons l'appel de routine à distance à la section 2.2 page 51. Nous nous intéresserons en particulier à RMI [145] qui permet d'abstraire les communications sous la forme d'appels de méthode à distance. Un tel appel implique le transfert d'objets entre machines virtuelles distinctes. Nous étudierons ensuite les problèmes que posent des approches qui franchissent une étape supplémentaire (comme celle de la plate-forme JavaParty [67]) en masquant l'appel de méthode à distance derrière la syntaxe d'un appel local. Nous verrons au chapitre 3 que JToe propose une abstraction des communications sous la forme de transfert d'objets mais que ce transfert est rendu explicite afin d'éviter de tels problèmes.

Tout comme pour les arguments dans le cas de l'appel de routine à distance, l'envoi d'un objet avec JToe se fait à l'initiative de l'émetteur. Cependant, la notification de l'arrivée d'un nouvel objet et son intégration au programme en cours d'exécution se fait, sur le récepteur, grâce à un mécanisme événementiel : une routine fournie par l'utilisateur à la couche JToe est appelée pour indiquer l'arrivée du nouvel objet. En cela, JToe se rapproche des plates-formes à messages actifs que nous présentons dans la section suivante.

2.1 Messages actifs

Le principe des messages actifs [137] est simple : un message actif contient dans son en-tête l'adresse d'une séquence d'instructions – appelée *handler* – qui sera exécutée à la réception du message. A la différence d'un appel de procédure à distance, l'objectif d'un *handler* n'est pas d'exécuter un ensemble de calculs complexes et de répondre à une requête. Au contraire, il s'agit d'être le plus bref possible pour intégrer les données contenues dans le message au sein du programme en cours d'exécution. En effet le *handler* d'un message actif est exécuté entièrement sans interruption, en conséquence la prise en charge de nouveaux messages n'est pas possible tant qu'un *handler* s'exécute.

D'un certain point de vue, on peut considérer les messages actifs comme une généralisation des fonctionnalités déjà offertes par le matériel et le système. Au lieu d'exécuter toujours le même code privilégié lorsque des données arrivent sur une interface réseau, on exécutera un code utilisateur qui dépendra directement du message reçu.

L'interface *Generic Active Message Interface* [36] a été proposée pour les messages actifs. Celle-ci définie cinq opérations de base :

- `request`
  Permet d'envoyer un message actif. Le *handler* de ce message est précisé ainsi que les arguments pour ce *handler*. Au maximum quatre mots de quatre octets peuvent être passés en argument.
- `reply`
  Permet d'envoyer une réponse à une requête. Seul un *handler* peut envoyer une réponse et il ne peut en envoyer qu'une seule. De plus, cette réponse doit être adressée à l'émetteur de la requête.
- `store`
  Permet de copier une zone mémoire à une adresse donnée dans la mémoire d'un nœud distant. Cette opération reçoit en argument l'adresse d'un *handler* sur le nœud distant. Ce dernier est exécuté lorsque toutes les données ont été reçues.
- `get`
  Permet de recopier une zone mémoire distante dans la mémoire locale. Cette opération reçoit en argument l'adresse d'un *handler* local qui sera appelé lorsque toutes les données auront été rapatriées localement.
- `poll`
  Permet de demander explicitement le traitement de messages reçu mais non encore pris en compte.

Cette interface doit permettre l'implantation de routines de communication classiques comme MPI. Cependant, certaines difficultés sont rapidement rencontrées. La routine MPI `MPI_Send` permet de transmettre un message vers un nœud distant. Son implantation exploite la primitive `store`. Comme `store` doit connaître, à l'avance, l'adresse à laquelle placer les données sur la machine destinataire, elle présuppose une phase de

négociation entre l'émetteur et le récepteur afin que ce dernier transmette au premier une adresse valide [23].

La bibliothèque LAPI (Low Level Application Programming Interface) [115] résout ce problème en adoptant une approche légèrement différente. On y retrouve le même type de primitives que celles vues précédemment : `Amsend` pour l'envoi d'un message actif et `Put` et `Get` pour la copie de données, respectivement, vers un nœud distant ou depuis un nœud distant. LAPI diffère par les types de messages actifs qui peuvent être transmis et par leur gestion.

Un message actif peut être d'une taille quelconque et peut donc être composé de plusieurs paquets réseau. Lors de la réception du premier paquet d'un message actif, son *handler* est exécuté. Son rôle est de fournir à la bibliothèque LAPI une adresse mémoire à laquelle les données du message entrant devront être stockées ainsi, éventuellement, qu'un pointeur vers une séquence d'instructions – appelée *completion handler* – qui sera appelée une fois le message totalement reçu. Ainsi, pour revenir à la primitive `MPI_Send`, l'émetteur d'une donnée n'a plus besoin de décréter le futur emplacement des données qu'il transmet. La bibliothèque LAPI fournit également des primitives de synchronisation entre processus. Elle est disponible sous IBM/SP.

Un problème des mécanismes à messages actifs est celui de garantir qu'un *handler* ne va pas bloquer. Si cela se produit, tout le système est bloqué, c'est donc une contrainte cruciale. Différentes propositions ont été faites, comme l'utilisation d'un *pool* de processus légers qui permet de prendre en charge les *handlers* effectuant une opération bloquante ou dépassant un certain temps d'exécution.

Par ailleurs, l'adresse de la routine contenue dans l'en-tête d'un message actif permet à l'émetteur de demander l'exécution d'un code quelconque au récepteur. Il faut donc, d'une part assurer que l'adresse correspond effectivement à une routine et pas à une zone mémoire quelconque et, d'autre part vérifier que cette routine peut être exécutée sans risque via un message actif. Même si JToe peut, d'un certain point de vue, se comparer aux messages actifs, nous verrons au chapitre suivant que le code exécuté lors de la réception d'un message n'est pas spécifié par l'émetteur du message mais par le récepteur via un mécanisme évenementiel. Ainsi, le problème d'un message demandant l'exécution de code situé à une adresse mémoire quelconque ne se pose pas.

2.2 Appel de *routine* à distance

Dans cette section, nous allons étudier les mécanismes d'appel de *routine* à distance. Nous allons voir l'appel de procédure à distance ainsi que l'appel de méthode à distance avant d'aborder les problèmes qui peuvent être liés au masquage syntaxique des communications.

2.2.1 Appel de procédure à distance

L'appel de procédure permet le transfert de contrôle et de données entre deux procédures présentes sur une même machine. L'appel de procédure à distance [12] étend cette notion à deux procédures situées sur deux machines éventuellement distinctes. Lorsqu'un appel est fait à une procédure distante, l'appelant est suspendu, les arguments de la procédure sont transférés, via le réseau, sur la machine où la procédure doit s'exécuter. Sur la machine de l'appelé, la procédure souhaitée est exécutée avec les arguments reçus par le réseau. Lorsque cette procédure se termine, le résultat de celle-ci est transféré, via le réseau, sur la machine de l'appelant. L'appelant redémarre en récupérant le résultat de la procédure appelée comme si celle-ci s'était exécutée localement.[1]

L'un des protocoles les plus répandus pour l'appel de procédure à distance est ONC RPC [119] – Open Network Computing Remote Procedure Call – aussi connu sous le nom de *Sun RPC*. Il repose sur le langage RPC [119] qui définit des notions de programme et de version et permet de décrire un service RPC, c'est-à-dire un ensemble de procédures pouvant être appelées à distance. Les programmes 2.1, 2.2 et 2.3 page ci-contre présentent un exemple de code pour le développement d'un service RPC *hello world* en C, affichant `Hello World !` sur le serveur.

```
1 program HELLO_WORLD{
2   version HELLO_FIRST{
3     int SAY_HELLO(void) = 1;
4   } = 1;
5 } = 0x20000001;
```

Programme 2.1: hello.x

```
1 #include <rpc/rpc.h>
2 #include "hello.h"
3
4 int *say_hello_1_svc(void *arg, struct svc_req *rqstp){
5   static int result;
6   result = printf("\nHello World !\n");
7   return &result;
8 }
```

Programme 2.2: server.c

Le programme 2.1 présente le source du fichier `hello.x` décrivant, dans le langage RPC l'interface du service *hello world*. Chaque composante du fichier `hello.x` se voit affecter une valeur. Par exemple ici, la procédure `SAY_HELLO` a l'identifiant `1` dans la version numéro `1` – également nommée `HELLO_FIRST` – du programme `HELLO_WORLD` qui lui-même a pour identifiant`0x20000001`. Le programme 2.2 décrit une implantation possible de ce service. Seule la procédure `say_hello_1_svc()`

[1] On notera cependant qu'un appel de procédure à distance peut être asynchrone. Nous verrons cela plus en détail dans le chapitre 4.

```
1  #include <stdio.h>
2  #include <rpc/rpc.h>
3  #include "hello.h"
4
5  main(int argc, char **argv)
6  {
7    CLIENT *clnt;
8    int *result;
9    char *server = argv[1];
10
11   clnt = clnt_create(server, HELLO_WORLD, HELLO_FIRST, "tcp");
12   if (clnt == NULL){
13     clnt_pcreateerror(server);
14     exit(1);
15   }
16
17   result = say_hello_1(NULL, clnt);
18   if (result == NULL){
19     clnt_perror(clnt, server);
20     exit(1);
21   }
22
23   if(*result < 0){
24     printf("\nError while printing on the server\n");
25   }else{
26     printf("\nHello World printed on server\n");
27   }
28
29   clnt_destroy(clnt);
30   exit(0);
31 }
```

Programme 2.3: client.c

doit être implantée. L'outil *rpcgen* génère l'ensemble des fichiers permettant le développement puis l'exécution du service, dont le fichier `hello.h`. Pour le client, un service est identifié par une variable de type `CLIENT*`. Pour obtenir une référence sur un service donné, le client indique le nom de la machine sur laquelle réside le service ainsi que l'identifiant du programme et le numéro de version souhaité (ligne 11 du programme 2.3 page précédente).

L'appel de procédure à distance tel que proposé dans ONC RPC et son utilisation, par exemple depuis le langage C, permettent d'abstraire les communications réseau en exploitant une syntaxe habituelle d'appel de procédure. Cependant, le fait que cette procédure soit distante apparaît clairement dans le programme. Le serveur n'implante pas une procédure `say_hello` mais une procédure `say_hello_1_svc`. De plus, au lieu de ne recevoir aucun argument, cette procédure en reçoit deux, le premier devant, dans ce cas,[2] être ignoré, le second permettant, si besoin est, d'interagir avec l'environnement d'exécution RPC. De plus, une procédure RPC ne reçoit pas directement les arguments déclarés dans l'interface RPC mais des pointeurs sur ces arguments et ne renvoie pas directement son résultat mais un pointeur sur ce dernier (ligne 7 du programme 2.2 page 52).

L'appel d'une procédure à distance n'est pas non plus transparent pour le client. La procédure à appeler ne se nomme pas, dans notre cas, `say_hello` mais `say_hello_1`. Le résultat de la procédure n'est pas celui attendu d'après la spécification RPC du service mais un pointeur sur ce dernier. Enfin, la procédure reçoit un argument supplémentaire qui identifie le service RPC auquel on s'adresse (ligne 17 du programme 2.3 page précédente).

On peut noter que l'utilisation de versions dans le langage RPC permet de faire coexister plusieurs versions d'un même service simultanément. Le nouveau service RPC `HELLO_WORLD` défini dans le programme 2.4 est compatible avec la version `HELLO_FIRST` et peut donc être utilisé par le client défini dans le programme 2.3 page précédente.

```
1 program HELLO_WORLD{
2   version HELLO_FIRST{
3     int SAY_HELLO(void) = 1;
4   } = 1;
5   version HELLO_SECOND{
6     int SAY_HELLO(void) = 1;
7     int SAY_GOODBYE(string name) = 2;
8   } = 2;
9 } = 0x20000001;
```

Programme 2.4: hello.x version 2

[2] Il s'agit en fait des arguments de la procédure déclarés dans `hello.x`. Dans notre cas, nous avons déclaré un argument `void` (pas d'arguments) et cela se traduit par la présence d'un argument de type `void*`.

Chaque procédure RPC s'applique à un service donné, identifié par l'argument `CLIENT*`. De plus, chaque service peut proposer une implantation distincte des procédures. De ce fait, on peut considérer que la programmation RPC n'est pas strictement procédurale mais se rapproche, par certains aspects, de la programmation objet, `CLIENT*` désignant l'objet cible de l'appel.

ONC RPC abstrait les communications liées aux différents espaces mémoire mis en jeu dans le cadre d'un environnement distribué sous la forme classique d'un appel de procédure. Cependant, ONC RPC ne masque pas ces communications. La différence syntaxique entre un appel de type RPC et un appel de procédure local attire l'attention du développeur sur les autres différences à prendre en compte.

2.2.2 Appel de méthode à distance

L'appel de procédure à distance permet d'exploiter une procédure distante. L'appel de méthode à distance permet d'exploiter un objet distant et d'invoquer une méthode sur cet objet. Pour cela, la notion de référence sur un objet est étendue au cadre distant, c'est-à-dire qu'une référence peut représenter un objet situé, en réalité, sur une machine distante. L'appel d'une méthode sur cette référence provoquera l'appel de la méthode en question à distance. Le procédé généralement utilisé est celui du *stub*. C'est-à-dire qu'une référence sur un objet distant est en fait une référence sur un objet local, un *stub*, dont le rôle est de transférer tous les appels de méthode qu'il reçoit vers l'objet distant qu'il représente.

Le fonctionnement, dans son principe, est finalement très proche de l'appel de procédure à distance. Nous avons vu que l'exploitation des RPC pouvait s'apparenter à une certaine forme de programmation objet. L'appel de méthode à distance concrétise cela par l'exploitation *native* du paradigme objet.

Il existe différentes plates-formes d'appel de méthode à distance, nous allons en présenter deux : CORBA [62] et Java RMI [145].

CORBA

CORBA [62] – Common Object Request Broker Architecture – est une spécification proposée par l'Object Management Group, une organisation composée de plus de 600 membres [61]. L'objectif de CORBA est de fournir un standard pour la gestion d'objets distribués.

CORBA permet l'appel de méthode à distance. Parmi les caractéristiques de CORBA on peut citer le langage IDL qui permet de spécifier l'interface d'un objet, indépendamment du langage dans lequel celui-ci est implanté ou utilisé. Le langage IDL est en fait le pendant du langage RPC vu précédemment mais il offre, en outre, l'héritage d'interface. CORBA spécifie également de nombreux services : nommage, événements, persistance, etc [58]. Cependant, pour être conforme à la spécification CORBA, une plate-forme doit

simplement implanter la spécification CORBA-Core [63], c'est-à-dire le principe de l'appel de méthode à distance tel que défini dans CORBA ainsi qu'une correspondance du langage IDL vers au moins un langage de programmation [59].

```
1  module hello {
2      interface Hello {
3          void sayHello();
4      };
5  };
```

Programme 2.5: Hello.idl

```
1  import hello.*;
2
3  public class HelloImpl extends HelloPOA{
4      public void sayHello(){
5          System.out.println("Hello World !");
6      }
7  }
```

Programme 2.6: HelloImpl.java

```
14     public static void callSayHello(Hello hello){
15         hello.sayHello();
16     }
```

Programme 2.7: Extrait de Client.java

Le programme 2.5 présente la version IDL de notre service *hello world*. Par rapport à la version RPC de ce service, on constate que la notion de module est apparue, le mot clé `program` est remplacé par `interface` et la notion de version a disparu.

Nous avons choisi d'implanter ce service en Java. Le programme 2.6 montre cette implantation. Du fait de la correspondance IDL vers Java définie par l'OMG [60], il est nécessaire de définir une classe héritant de `HelloPOA` – générée par les outils de traduction IDL vers Java – et implantant la méthode `sayHello()`. En ce qui concerne le client, dont un extrait est présenté dans le programme 2.7, on peut constater que l'appel de méthode ne présente aucune différence syntaxique avec un appel de méthode qui aurait lieu sur un objet local. On peut d'ailleurs facilement imaginer une implantation locale de l'interface `hello.Hello`.

Ainsi, dans cet exemple, et à la différence de ONC RPC, appels locaux et distants ont la même syntaxe. Cependant, les interfaces sur lesquelles les appels de méthode sont faits sont forcément issues d'interfaces définies dans le langage IDL. Ainsi, un développeur ne peut pas invoquer les méthodes d'une interface qui avait été prévue, au départ, pour être locale et qui serait rendue distante au travers de CORBA.[3]

[3] Il est possible d'outrepasser cela mais on quitte alors le modèle de développement proposé par l'OMG.

Java RMI

L'une des forces de CORBA est de permettre l'appel de méthode à distance indépendamment des architectures et des langages utilisés pour implanter appelant et appelé. Nombre de langages bénéficient d'une traduction du langage IDL standardisée par l'OMG [59]. Java RMI [145] permet également l'appel de méthode à distance mais ne permet que l'interaction entre objets Java.

L'un des apports principaux de Java RMI [141] se situe dans l'exploitation complète du paradigme objet et en particulier du polymorphisme et de la sélection dynamique de méthode. Une des caractéristiques du polymorphisme dans les langages orientés objet est de permettre le passage d'arguments spécialisés à une méthode qui attend un type plus générique comme, par exemple, dans le programme suivant :

```
1 ...
2 Vector x = new Vector();
3 x.add(...);
4 ...
5 System.out.println(x);
6 ...
```

La méthode `println()` attend un argument du type générique `Object`. On lui passe un argument de type `Vector` et cela fonctionne car `Vector` est une sous-classe de `Object`. Cette méthode `println()` a recours à la méthode `toString()` de l'objet reçu en argument. Ici, la sélection dynamique de méthode va permettre que ce soit la méthode `toString()` redéfinie dans la classe `Vector` qui soit appelée et non celle de la classe `Object`. Cela aura pour effet de produire un affichage adapté à un vecteur.

Si on suppose maintenant que l'instruction de la ligne 5 constitue un appel de méthode à distance – autrement dit, si on suppose que `System.out` est un objet distant – alors les choses se compliquent. En effet, `x` est passé par copie vers la machine de l'objet serveur mais il n'y a aucune raison pour que la classe `Vector` existe sur cette machine.[4]

Pour permettre un véritable polymorphisme dans le passage des arguments, RMI exploite le chargement dynamiquement de classe disponible en Java. Comme on peut le voir sur la figure 5 page suivante, lorsque l'objet `x` est reçu sur le serveur, la classe réelle de cet objet est téléchargée via une URL (étape 2). Cette URL est le *codebase* de l'objet et a été transmise par la plate-forme RMI en même temps que l'objet. Une fois la classe chargée (étape 3), la copie de l'objet `x` – notée `x'` sur la figure – peut être initialisée (étape 4). Enfin, l'appel de méthode peut être effectivement réalisé (étape 5). Le transfert d'objets s'effectue grâce au mécanisme de *serialization* proposé en Java. La *serialization* a un impact important sur les performances de RMI et différents travaux proposent une

[4] Pour plus de clarté, nous avons choisi d'utiliser un vecteur, instance d'une classe Vector, dans notre exemple. La classe `java.util.Vector` est disponible de façon standard dans le jdk et donc dans le cas d'un client et d'un serveur tous les deux écrits en Java, le problème que nous présentons ne se pose pas. Cependant, dans le cas général il n'y a aucune raison pour que cette classe soit présente sur les deux sites.

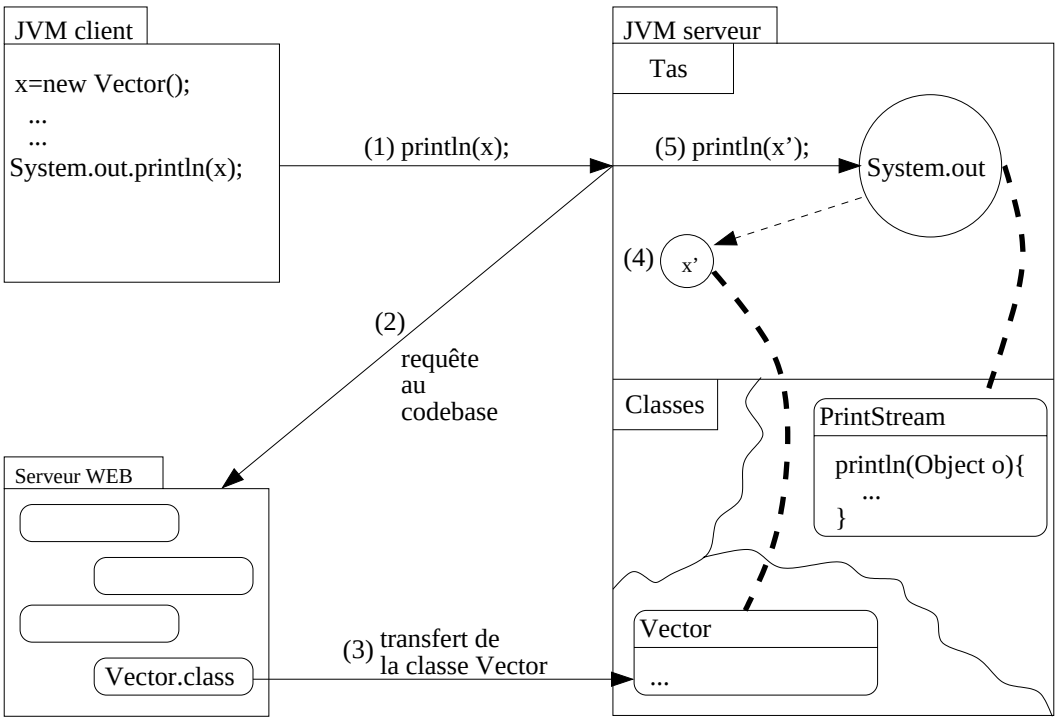


FIG. 5 – Le chargement dynamique de classe avec Java RMI.

alternative dans le cadre de l'appel de méthode à distance. Nous reviedrons sur cela au chapitre suivant consacré à JToe.

Le chargement dynamique de classe est un mécanisme de base de la plate-forme Java. La possibilité de télécharger un code dynamiquement via une URL et de l'exécuter introduit des risques en terme de sécurité. L'appel d'une méthode de la classe nouvellement téléchargée peut ouvrir la porte à des abus et des comportements de type *cheval de Troie*. La gestion de la sécurité dans RMI repose sur celle déjà existante au sein de la plate-forme Java et que nous avons décrite dans la section 1.3.1 page 18. On notera d'ailleurs sur la figure 5 que la classe `Vector` n'aura pas les même droits que d'autres qui n'auraient pas été chargées dynamiquement. Cela est représenté par une partition de l'espace des classes de la `JVM serveur` sur le schéma.

```
1 public interface Hello extends java.rmi.Remote{
2     public void sayHello() throws java.rmi.RemoteException;
3 }
```

Programme 2.8: Hello.java

Le programme 2.8 présente le code de l'interface RMI de notre service *hello world*. C'est le pendant des programmes 2.1 page 52 et 2.5 page 56 définissant les interfaces du service *hello world* respectivement dans les langages RPC et IDL. RMI étant limité au langage Java, il s'agit directement d'une interface Java. On constate que

celle-ci hérite de l'interface `java.rmi.Remote` (cf. ligne 1) et que chaque méthode – ici il n'y en a qu'une – déclare potentiellement lever l'exception contrôlée `java.rmi.RemoteException` (cf. ligne 2).

```
1  import java.rmi.*;
2  import java.rmi.server.UnicastRemoteObject;
3
4  public class HelloImpl extends UnicastRemoteObject implements Hello{
```

```
10     public void sayHello() throws RemoteException{
11         System.out.println("Hello World !");
12     }
13 }
```

Programme 2.9: HelloImpl.java

Une interface distante RMI doit respecter deux contraintes : hériter de l'interface `java.rmi.Remote` et déclarer l'exception `java.rmi.RemoteException` pour chacune de ses méthodes. Ces deux critères sont vérifiés par le compilateur *rmic* chargé de générer les différents composants nécessaires à la plate-forme RMI. L'implantation d'une interface RMI se fait ensuite de façon naturelle par rapport au langage Java, c'est-à-dire qu'il suffit de définir une classe implantant cette interface. C'est le cas de la classe `HelloImpl` dont un extrait est présenté sur le programme 2.9.

```
9      public static void callSayHello(Hello hello){
10         try{
11             hello.sayHello();
12         }catch(RemoteException e){
13             System.err.println("Problème réseau");
14             e.printStackTrace();
15         }
16     }
17 }
```

Programme 2.10: Extrait de Client.java

Le programme 2.10 est un extrait du code du client du service *hello world*. La méthode `callSayHello()` reçoit en argument un objet de type `Hello` et invoque sa méthode `sayHello()` comme dans la version CORBA du programme 2.7 page 56. La différence est ici le fait que le client est obligé de gérer l'exception contrôlée `java.rmi.RemoteException`. Cela fait apparaître au niveau syntaxique l'aspect distant de cet appel.

L'héritage de l'interface `Remote` ainsi que la déclaration de l'exception `RemoteException` assurent, comme pour CORBA, qu'on ne peut pas directement accéder à distance à une interface ou une classe non prévue pour RMI. Inversement, il n'est pas censé exister de version locale d'une méthode RMI. Par rapport à CORBA, RMI assure une visibilité syntaxique de l'aspect distant par la gestion de l'exception `RemoteException` qui est imposée.

2.2.3 Masquage syntaxique des communications

Nous avons présenté ONC RPC, CORBA et Java RMI. Ces trois plates-formes permettent l'appel de routine à distance, qu'il s'agisse de procédures ou de méthodes. La syntaxe utilisée est celle d'un appel classique. Cependant, dans la plupart des cas, certains éléments attirent l'attention du programmeur sur l'aspect distant de son appel : argument de type `CLIENT*` dans ONC RPC, déclaration de l'exception `java.rmi.RemoteException` dans Java RMI. Dans tous les cas, il n'est pas possible – ou pas conforme à la démarche de programmation définie – d'implanter de manière distante une interface locale déjà existante. Ainsi, bien qu'il pourrait être intéressant, par exemple dans le cas du langage Java, de permettre l'accès à une collection distante en utilisant les méthodes standard de l'interface Java `java.util.Collection`, cela est impossible avec les plates-formes que nous venons de décrire. D'autres autorisent cela, ce qui a pour effet de masquer à l'appelant l'aspect distant de son appel. C'est le cas, par exemple, de la plate-forme JavaParty [67, 103].

JavaParty offre des primitives pour la programmation Java sur *cluster*. Cette plate-forme permet, par exemple, la création d'un objet sur un nœud donné d'un *cluster* et propose diverses primitives de placement. Elle permet l'appel de méthode à distance syntaxiquement transparent. JavaParty repose sur Java RMI ou KaRMI (cf. section 3.2.2 page 71) pour les communications. Quand RMI nécessite la définition d'une interface distante et la levée d'exceptions spécifiques pour l'écriture d'une classe distante, JavaParty se contente d'ajouter un mot clé au langage : `remote`. Toutes les méthodes d'une classe déclarée `remote` seront accessibles à distance. On retrouve également ce type d'approche dans la plateforme Do ![83] qui propose un modèle de programmation parallèle à base de collections distribuées dans lequel l'implantation, par une classe, de l'interface spéciale `Accessible` rend toutes les méthodes de cette classe accessibles à distance.

```
1 package hello;
2
3 public remote class Hello{
4     public void sayHello(){
5         System.out.println("Hello World !");
6     }
7 }
```

Programme 2.11: Hello.java

Le programme 2.11 présente la définition de notre service *hello world* avec JavaParty. Il s'agit d'une classe normale dont la méthode `sayHello()` affiche `Hello World!`. Le mot clé `remote` de la ligne 3 indique au pré-compilateur de JavaParty que cette classe est accessible à distance. On peut noter qu'il est possible – même si nous n'exploitons pas cette possibilité dans le programme présenté – d'implanter n'importe quelle interface existante et de rendre ainsi ses méthodes directement accessibles à distance de façon transparente.[5]

[5]En fait, les méthodes `final` sont une exception.

```
1  import hello.Hello;
2
3  public class ClientServer{
4      public static void main(String[] args){
5          Hello hello = new Hello(jp.lang.Node.getNode(1));
6          callSayHello(hello);
7      }
8
9      public static void callSayHello(Hello hello){
10         hello.sayHello();
11     }
12 }
```

Programme 2.12: ClientServer.java

Pour exploiter ce service, il suffit de créer une instance de notre classe `Hello` grâce à l'opérateur `new` puis d'invoquer sa méthode `sayHello()`. Le code de la classe `ClientServer` est présenté dans le programme 2.12. On peut constater qu'on a utilisé un constructeur de la classe `Hello` généré par le compilateur JavaParty qui nous permet de spécifier le nœud sur lequel on souhaite que l'instance soit créée – ici le nœud `1`.

Sur l'exemple *hello world*, on constate qu'au niveau de la méthode `callSayHello()` du programme 2.12 aucun élément syntaxique n'indique que l'appel à la méthode `sayHello()` est distant. De plus, et à la différence de ce que nous avons vu avec CORBA, une classe *remote* peut implanter n'importe quelle interface locale existante. Nous allons insister sur le problème que peut poser cette transparence syntaxique au travers d'un exemple un peu plus complexe que *hello world*.

```
1  package xobservable;
2
3  public interface XObservable{
4      public void addXObserver(XObserver o);
5  }
```

```
1  package xobservable;
2
3  public interface XObserver{
4      public void update(XObservable o, int newXValue);
5  }
```

Programme 2.13: Les deux interfaces du paquetage `xobervable`.

Considérons un paquetage définissant la notion de *xobservation* et présenté dans le programme 2.13. On supposera que ce paquetage existe et qu'il est disponible, pourquoi pas dans le JDK. Un objet de type `XObservable` possède une propriété `x` dont les changements d'état peuvent être observés. Par souci de simplicité, nous avons décidé que la valeur de cette propriété `x` serait de type entier (`int`). Un objet de type `XObserver` est capable de *xobserver* un `XObservable`.

Ces deux interfaces sont prévues, en tout cas syntaxiquement, pour être exploitées de façon locale. Le programme 2.14 présente le source de la classe `PrintXObserver` qui implante l'interface `XObserver` de façon simple en affichant un message dans la console lorsque l'état du *xobservé* change. De plus, elle propose une méthode *static* `setObserver()` qui enregistre une instance de cette classe auprès d'un `XObservable`. Un exemple trivial d'application locale utilisant cette classe est présenté à titre d'information programme 2.18 page 65.

```
1  import xobservable.*;
2
3  public class PrintXObserver implements XObserver{
4      public void update(XObservable o, int newXValue){
5          System.out.println("Nouvelle valeur de x pour "+
6                             o + " : " + newXValue);
7      }
8      public static void setObserver(XObservable o){
9          o.addXObserver(new PrintXObserver());
10     }
11 }
12
```

Programme 2.14: La classe `PrintXObserver` affiche le changement d'état et la nouvelle valeur.

On souhaite maintenant utiliser JavaParty pour définir une classe distante implantant l'interface `XObservable`. Le programme 2.15 page suivante propose un exemple d'une telle implantation. Puisque cette classe implante l'interface `XObservable`, elle peut directement être utilisée par la classe `PrintXObserver` du programme 2.14. On peut donc écrire l'application présentée programme 2.16 page suivante.

Dans cette application, le code de la classe `PrintXObserver` n'a absolument pas changé par rapport à une application locale. En particulier, l'enregistrement de l'observateur auprès de l'observé qui est réalisé par la méthode `setObserver()` du programme 2.14, n'a absolument pas changé. Cependant, le comportement sera totalement différent puisque cette dernière méthode n'aura pas pour effet de simplement enregistrer l'observateur auprès de l'observé mais bel et bien d'effectuer une copie de l'observateur sur la machine de l'observé et d'enregistrer cette copie. Cela se manifestera, dans cette application, par un affichage dans la mauvaise console mais les conséquences peuvent être plus problématiques si on imagine que l'observateur manipule une structure de données complexe.

On peut considérer que c'est à la charge de celui qui enregistre le *xobservateur* de s'assurer de l'aspect local ou distant de l'appel et de la compatibilité du *xobservateur* avec un éventuel aspect distant. Cependant, dans l'exemple présenté, le code qui enregistre le *xobservateur* (la méthode `setObserver()` du programme 2.14) n'a pas été modifié et il n'y a pas de moyen syntaxique de savoir que l'objet fourni est distant. Cela devient alors du rôle de celui qui crée l'objet distant et le fournit à cette méthode de s'assurer que cette méthode ainsi que tous les intermédiaires éventuels sont bien compatibles avec cet aspect

```
1  import xobservable.*;
2
3  public remote class JavaPartyXObservable implements XObservable{
4      private int x = 0;
5
6      public void setX(int value){
7          this.x = value;
8          notifyObservers();
9      }
10
11     private java.util.Collection observers = new java.util.HashSet();
12     public void addXObserver(XObserver o){
13         observers.add(o);
14     }
15
16     private void notifyObservers(){
17         for(java.util.Iterator it = observers.iterator();
18             it.hasNext(); ){
19             ((XObserver)it.next()).update(this, x);
20         }
21     }
22 }
```

Programme 2.15: Une classe JavaParty `remote` implantant l'interface `XObservable`.

```
1  import xobservable.*;
2
3  public class AppJavaParty{
4      public static void main(String[] args){
5            JavaPartyXObservable jo = new JavaPartyXObservable();
6            PrintXObserver.setObserver(jo);
7            jo.setX(12043);
8            jo.setX(3204075);
9            jo.setX(0);
10     }
11 }
```

Programme 2.16: La classe `AppJavaParty` construisant un `JavaPartyXObservable` avant de le fournir à la classe existante `PrintXObserver`.

distant. C'est justement le rôle d'un système de typage que de définir, pour une méthode donnée, l'ensemble des arguments qu'elle peut accepter. Masquer syntaxiquement l'aspect distant d'un objet sans prendre en charge les problèmes de sémantique équivaut alors à violer ce système de typage et le rend inutile, pour faire reposer tout ce travail sur les épaules du développeur.

2.3 Transfert d'objets en Java

L'appel de méthode à distance nécessite le transfert d'arguments vers la machine de l'appelé. Nous avons vu que RMI met en place un tel transfert pour les objets Java, de même pour JavaParty et d'autres plate-formes que nous n'avons pas évoquées ici. Dans le cas de JavaParty, l'absence totale de frontière entre local et distant, sur le plan de la syntaxe, aboutit à des difficultés de programmation que nous avons mises en évidence.

RMI pose moins ce problème puisque nous avons vu que des éléments syntaxiques permettent au développeur d'avoir conscience de l'aspect distant de son appel. Néanmoins, on doit noter que le passage d'arguments avec RMI ne s'effectue pas toujours par copie. Le choix de passer un argument par copie ou par référence dépend de la nature de l'argument et de son état. Un objet distant, c'est-à-dire un objet dont les méthodes peuvent être invoquées à distance, ne sera pas passé par copie mais par référence. Cependant, dans le cas où cet objet distant ne serait pas encore *exporté*, c'est-à-dire s'il n'était pas encore prêt à recevoir des appels de méthode, alors il serait passé par copie.

Même lorsque la syntaxe indique clairement qu'une opération – un appel de méthode dans le cas de RMI – est distante, la nature de la transmission des données de cette opération – passage de paramètre par copie ou par référence dans le cas de RMI – n'est pas forcément claire. Comme nous le verrons au chapitre suivant, l'API JToe clarifie les choses en proposant la méthode explicite `copy` pour le transfert d'objets.

Dans le cas de l'appel de méthode à distance, le transfert de données se fait, en ce qui concerne les arguments, à l'initiative de l'émetteur. C'est le cas également dans la bibliothèque JToe. Cependant, nous verrons que celle-ci se limite au transfert d'objets, se rapprochant ainsi des messages actifs. Cette API peut être utilisée pour implanter des mécanismes d'appel de routine à distance ou tout autre paradigme impliquant le transfert d'objets. La limitation au *simple* transfert d'objets permet de simplifier son implantation en évacuant vers les couches de plus haut niveau toutes les difficultés que peuvent être l'écriture d'un ramasse-miettes distribué ou la gestion des processus légers pour l'exécution des méthodes. Ceci permet d'isoler les problématiques et de les traiter indépendamment les unes des autres. Nous présentons JToe au chapitre suivant.

```
1  import xobservable.*;
2
3  public class App{
4      public static void main(String[] args){
5          LocalXObservable o = new LocalXObservable();
6          PrintXObserver.setObserver(o);
7          o.setX(12043);
8          o.setX(3204075);
9          o.setX(0);
```

```
1  import xobservable.*;
2
3  public class LocalXObservable implements XObservable{
4      private int x = 0;
5
6      public void setX(int value){
7          this.x = value;
8          notifyXObservers();
9      }
10
11     private java.util.Collection xobservers = new java.util.HashSet();
12     public void addXObserver(XObserver o){
13         xobservers.add(o);
14     }
15
16     private void notifyXObservers(){
17         for(java.util.Iterator it = xobservers.iterator();
18             it.hasNext(); ){
19             ((XObserver)it.next()).update(this, x);
20         }
21     }
22 }
```

Programme 2.18: Une implantation locale de l'interface `XObservable`.

Chapitre 3

Contribution à la prise en charge de la mémoire répartie : JToe

L'API JToe [28] (ou Transfert d'objets efficace en Java) s'intéresse, comme son nom l'indique, au transfert d'objets Java sur un réseau. Le transfert d'objets est un enjeu essentiel pour le calcul distribué en Java. Nous avons déjà évoqué au chapitre précédent son utilisation par RMI [78], par exemple. Différents travaux [13, 14, 87, 100] se consacrent à l'amélioration des performances de ce processus et la bibliothèque JToe s'inscrit parmi ceux-là.

L'objectif de JToe est de spécifier un mécanisme de transfert d'objets permettant des implantations efficaces. Pour cela, cette API se concentre **uniquement** sur le transfert d'objets, elle cherche à n'imposer **aucune limitation** dans les possibilités d'implantation et est **minimale** : seules deux interfaces et deux méthodes sont définies, ce qui la rend facile à comprendre et à implanter.

En Java, la *serialization* constitue un élément de base standard pour le transfert d'objets. Nous allons présenter ce mécanisme avant d'en étudier certaines alternatives. Nous présenterons ensuite l'API JToe qui, à la différence des autres travaux, se limite *uniquement* au transfert d'objets afin de simplifier l'implantation de ce mécanisme. Nous nous intéresserons ensuite en détail à une implantation de cette API sur JikesRVM [71].

3.1 La *Serialization* en Java

Le mécanisme dit de *serialization* permet de transformer un objet Java en une suite d'octets. Ce mécanisme a été introduit pour permettre le stockage d'objets sur des périphériques persistants et le transfert d'objets pour l'appel de méthodes à distance [108].

La *serialization* d'un objet se fait grâce à la méthode `writeObject()` de l'interface `ObjectOutput` et en particulier grâce à son implantation dans la classe

`ObjectOutputStream`. Pour cela, la classe de l'objet en question est analysée dynamiquement. Chaque champ de type primitif est encodé directement, chaque champ de type tableau ou objet est encodé de façon récursive grâce à la méthode `writeObject()`. Ainsi, l'objet et tous les objets qu'il référence sont encodés dans une suite d'octets. Il est ensuite possible de restaurer le graphe d'objets induit grâce à la méthode `readObject()` de l'interface `ObjectInput`. La méthode `annotateClass()` de `ObjectOutputStream` permet, en outre, d'ajouter toute information pertinente pour chacune des classes des objets transmis lors d'une *serialization* comme, par exemple, une URL à laquelle le code de cette classe est disponible.

Une classe peut implanter différentes méthodes pour modifier la façon dont la *serialization* de ses instances est effectuée. Dans le cas où une classe implante une méthode `writeObject()`, celle-ci sera invoquée lors de la *serialization* d'une instance de cette classe. Symétriquement, une méthode `readObject()` peut être définie pour la restauration d'un objet. Ces méthodes peuvent être définies pour des raisons de performance – encoder de façon compacte une structure de données de grande taille – ou pour garantir une certaine cohérence – gérer des structures qui dépendent du système d'exploitation local et qui ne doivent pas être passées telles quelles comme une *socket*.

Il existe encore d'autres façons d'interagir avec le mécanisme de *serialization* [91], comme les méthodes `writeReplace` et `readResolve` qui permettent de remplacer, à la volée, un objet par un autre. Enfin, lors de l'encodage d'un objet, le numéro de version de sa classe est encodé également. Si le code de cette classe ne précise aucun numéro de version, le numéro de version utilisé correspondra au calcul d'une empreinte de ce code. Ce numéro permet de s'assurer que la classe disponible lors de la restauration de l'objet est bien compatible avec celle utilisée lors de son encodage.

Le mécanisme de *serialization* de Java s'attache à fournir un degré de sécurité correspondant à celui déjà existant au moment de son introduction. En Java, une classe ne peut accéder aux champs privés d'une autre sans privilège particulier. Il ne s'agit pas ici, comme dans la plupart des langages, de vérifications du compilateur mais bel et bien de vérifications au moment de l'exécution. L'encodage sous la forme d'octets d'objets peut permettre à quelqu'un y ayant accès de connaître la valeur d'un champ privé qui lui est normalement inaccessible. Pour limiter cela, différentes règles s'appliquent. Par défaut, les instances d'une classe ne sont pas *serializables*. Seules celles dont la classe implante l'interface `Serializable` le sont. Les méthodes `writeObject()` et `readObject()` sont privées. Il est donc impossible de les invoquer pour obtenir des informations sur l'état d'un objet ou le modifier. Seul le mécanisme de *serialization* peut les exploiter.

Une classe peut également être *serializable* dans le cas où elle implante l'interface `Externalizable`. Elle devra alors définir les méthodes `readExternal()` et `writeExternal()`. Ces méthodes étant déclarées `public`, elles ne bénéficient pas des même protections que les méthodes spéciales `readObject()` et `writeObject()` et c'est au développeur de s'assurer qu'elles ne sont pas appelées de façon abusive.

Les champs sensibles peuvent être écartés de la *serialization* en les déclarant *transient* ou en définissant une méthode spéciale `writeObject()` ignorant ces champs. Il est également possible de définir la liste des champs devant être considérés lors de la *serialization* en déclarant un champ *serialPersistentFields* dont la valeur sera un tableau de type *ObjectStreamField* contenant l'identité de chacun des champs à considérer.

La *serialization* met également en place des mécanismes pour limiter l'impact d'une corruption des données utilisées pour la restauration d'un objet. Il est par exemple possible qu'une instance d'une sous-classe d'une classe donnée soit restaurée mais que les données de la *serialization* n'indiquent pas cette relation d'héritage. Dans ce cas, aucune donnée associée à la super-classe n'est disponible. Lorsque le mécanisme de *serialization* détecte l'absence de données pour l'initialisation d'un objet alors qu'elles devraient être présentes, la méthode `readObjectNoData()`, lorsqu'une classe la définit, est appelée. D'autre part, lorsqu'une classe possède un champ privé utilisé pour référencer un objet modifiable mais qu'elle utilise cet objet pour stocker un état qui ne doit pas pouvoir être modifié, il est alors possible de corrompre les données volontairement en ajoutant artificiellement une référence vers ce champ et permettre ainsi sa modification après restauration. La méthode `readUnshared()` de la classe `ObjectInputStream` permet justement de garantir qu'un objet restauré ne peut être référencé plusieurs fois.

Les méthodes `writeObject()` et `readObject()` des classes `ObjectOutputStream` et `ObjectInputStream` sont déclarées `final`. C'est-à-dire qu'il est impossible de les redéfinir. Cela garantit le comportement général de la *serialization* et notamment le fait que les méthodes spéciales `readObject()` et `writeObject()` d'une classe seront bien appelées.

Pour éviter qu'une méthode `readObject()` ne puisse lire plus de données que celles de l'objet qui la concernent – ce qui lui permettrait d'obtenir des informations sur d'autres objets du flux éventuellement sensibles – les données de chacun des objets sont séparées les unes des autres. Une exception est levée ou une fin de fichier signalée, selon la situation, lorsqu'une méthode `readObject()` tente d'accéder à des données qui ne la concernent pas.

Enfin, il est possible de vérifier, après restauration, la validité d'un objet en implantant l'interface `ObjectInputValidation`. Lorsqu'un objet est restauré, le mécanisme de *serialization* invoque chacun des vérificateurs – objet implantant `ObjectInputValidation` – enregistrés auprès de la classe `ObjectInputStream`. Si l'un d'entre eux considère le nouvel objet comme invalide, celui-ci sera rejeté.

3.2 Alternatives à la *serialization*

Le mécanisme de *serialization* que nous venons de décrire est généraliste et offre un certain nombre de garanties, en terme de sécurité notamment. Cependant, il est réputé peu performant. Christian Nester, Michael Philippsen et Bernhard Haumacher [100], par exemple, ont pu mettre en évidence des points critiques concernant les performances de RMI liés à la *serialization*. D'après leurs tests, la *serialization* représente plus de 25% du coût d'un appel de méthode avec RMI. Différentes alternatives ont été proposées pour améliorer les performance de la *serialization* [13, 14, 87, 100].

Il est possible de redéfinir la *serialization* en proposant une implantation distincte des interfaces `ObjectOutput` et `ObjectInput`.[1] D'un certain point de vue, on peut considérer l'interface définie par les classes `Object(Output/Input)Stream` comme une interface bas niveau car elle fait apparaître de façon explicite la notion de flux utilisée pour la *serialization*. Or, différentes bibliothèques de communication ne reposent pas sur la notion de flux [115, 138] et une API orientée flux n'est donc pas forcément bien adaptée à leur exploitation.

Nous allons présenter ici différentes alternatives au mécanisme de *serialization*.

3.2.1 Expresso

Expresso [34] permet le transfert d'objets Java entre machines virtuelles homogènes. Expresso repose sur la machine virtuelle Kaffe [126]. L'objectif d'Expresso est d'exploiter l'homogénéité des machines virtuelles mises en jeu pour n'appliquer aucune transformation à la représentation interne des objets et transférer ces derniers en n'effectuant aucune copie intermédiaire.[2]

Pour permettre le transfert d'objets en zéro copie, Expresso repose sur la notion de *cluster* qui désigne une zone mémoire de la machine virtuelle. La base du transfert d'objet dans Expresso n'est donc pas, comme avec la *serialization*, le graphe des objets référencés par un objet racine donné mais le *cluster*. Ainsi, transférer un objet consiste à transférer le *cluster* dont cet objet fait partie. Cet objet ainsi que tous les objets situés dans ce même *cluster* seront transférés. Cependant, toute référence sur des objets à l'extérieur du *cluster* est ignorée. Il est donc à la charge du développeur d'allouer ses objets dans les *clusters* appropriés afin d'assurer un transfert cohérent de ses données.

Cette notion de *cluster* permet le transfert efficace d'objets Java. Cependant elle est un peu radicale et n'offre aucune garantie sur l'état des objets après transfert si des références *traversent* les *clusters*. D'une certaine façon, ce mécanisme de *cluster* s'oppose aux principes d'abstraction de la mémoire présents en Java et est complexe à exploiter.

[1] En pratique, il est nécessaire d'hériter des classes `ObjectOutputStream` et `ObjectInputStream`. La méthode spéciale `readObject()`, par exemple, reçoit en argument un objet de type `ObjectInputStream`.

[2] Dans la suite, nous utiliserons le terme de *communication zéro copie* pour faire référence aux mécanismes de communication n'effectuant aucune copie intermédiaire des données à transmettre ou reçues.

3.2.2 KaRMI

Le projet JavaParty [103] (Cf. section 2.2.3 page 60) propose une plate-forme de calcul distribué en Java et repose sur RMI. Ayant identifié RMI, et en particulier le mécanisme de *serialization* et de communication, comme un point sensible dans les performances des applications JavaParty, l'équipe du projet JavaParty a mis au point KaRMI et la *UKA-Serialization* [100]. A quelques restrictions près, qui ne sont pas supposées concerner la majorité des applications, une application RMI existante peut utiliser KaRMI de façon transparente et bénéficier des optimisations qu'il propose.

Comme nous l'avons vu précédemment, *serializer* un objet en Java ne nécessite rien d'autre que l'implantation, par la classe de cet objet, de l'interface `java.io.Serializable`. Cette interface ne définit aucune méthode, c'est un simple marqueur. C'est donc le mécanisme de la *serialization* qui procède à un travail d'introspection dynamique pour retrouver l'ensemble des champs à *serializer*. Pour éviter cela, la *UKASerialization* impose la définition de méthodes de *serialization* pour chaque classe, qui permettent d'éviter cette introspection dynamique et doivent donc améliorer les performances. Enfin, un accès direct aux *buffers* sous-jacents à la *serialization* ainsi qu'une gestion spécifique de ces derniers permettent d'accélérer globalement le processus. Le principe de l'approche consiste donc à demander au développeur d'écrire lui-même le code de *serialization* pour augmenter les performances par rapport à un mécanisme automatique.

RMI utilise les *sockets* pour ses communications. L'interface de RMI exporte cette notion de *socket* et permet à l'utilisateur de modifier les *sockets* utilisées, par exemple pour intégrer le cryptage des informations. Cette notion de *socket* n'est pas toujours adaptée au calcul haute performance et, en particulier, à certaines interfaces de communications. C'est pourquoi KaRMI remplace cette notion de socket par une notion, plus abstraite, de *technologie*. Une technologie permet de communiquer vers un objet distant KaRMI. C'est ce module qui est en charge de l'interface avec le média de communication bas niveau. Plusieurs technologies peuvent coexister simultanément lors d'une exécution d'un programme JavaParty, chaque objet distant pouvant exploiter une technologie différente.

3.2.3 RMIX

Comme nous venons de le voir, KaRMI [100] mais également d'autres travaux, comme Manta [87] par exemple, tentent de résoudre les problèmes de performance de RMI liés au mécanisme de la *serialization*. De plus, le protocole de communication utilisé par RMI n'est pas interopérable : il ne permet la communication qu'avec d'autres applications utilisant Java RMI. RMI-IIOP [122] et JAX-RPC [120] sont deux exemples d'alternative à RMI et à son protocole de communication pour l'interfaçage, respectivement avec des composants CORBA [62] grâce au protocole IIOP et avec des services web grâce à SOAP/HTTP [140]. Tous ces travaux, qui cherchent à fournir une alternative à RMI, sont distincts les uns des autres et ne présentent aucune compatibilité.

L'objectif de RMIX [78, 79, 80, 146] est de proposer une plate-forme commune d'interopérabilité. Pour cela, RMIX définit un ensemble minimum de fonctionnalités que doit fournir un protocole d'appel de méthodes à distance et permet à une application de vérifier si un protocole propose des fonctionnalités optionnelles dont elle aurait besoin. Par exemple, tout protocole RMIX doit être capable de transférer, via un mécanisme de type *serialization*, des valeurs de type primitif, des chaînes de caractères, des tableaux contenant des éléments de type primitif ou des chaînes de caractères, des références distantes, et des instances de classes déclarées `final` dont tous les champs sont soit de type primitif, soit des chaînes de caractères et qui, en plus, proposent des méthodes pour accéder à ces champs pour lire leur valeur ou les modifier (`getX()`, `setX()`) conformément aux spécifications Java beans [90]. Cependant, un protocole peut proposer le transfert de tout objet *serializable* de façon optionnelle.

RMIX offre actuellement un support pour le protocole standard de Java RMI, SOAP ou encore RPC [119] et permet, par la définition d'un certain nombre d'interfaces, de développer des méthodes alternatives pour le transfert d'objet et l'appel de méthode à distance qui soient toutes exploitables dans le cadre d'une plate-forme commune. Pour pouvoir implanter de telles alternatives avec RMIX, il est nécessaire de s'intéresser, non seulement au problème du transfert d'objets – pour le passage de paramètres – mais également, dans le même temps, aux problèmes soulevés par l'invocation de méthode à distance : gestion des processus légers, gestion de références distantes, ramasse-miettes distribué, etc.

3.2.4 Ibis

Ibis [132] est un projet mené à l'université Vrije à Amsterdam. Son objectif est de fournir un environnement de développement pour le calcul haute performance sur la grille en Java[132]. Pour cela, Ibis s'articule autour de l'IPL [132], ou *Ibis Portability layer*. L'IPL est un ensemble d'interfaces Java définissant des mécanismes de communication et de nommage. Le projet Ibis représente une suite aux travaux menés par la même équipe autour du projet Manta [87, 134]. Ibis reprend, par exemple, le principe de génération de code de *serialization* : pour éviter d'avoir recours à l'introspection pour repérer l'ensemble des champs de chaque objet qu'il est nécessaire de *serializer*, un code de *serialization* est généré pour chaque classe. Le calcul des champs n'est donc effectué qu'une seule fois, au moment de la compilation. Cependant, à la différence de Manta qui reposait sur le paradigme d'appel de méthode à distance, Ibis propose l'envoi de messages. De plus, alors que Manta nécessitait de compiler totalement un code Java en code machine [134], Ibis cherche à conserver la portabilité proposée par Java : un programme utilisant l'IPL est portable et 100% Java dans la mesure où il existe des implantations 100% Java de l'IPL. Cependant, il est possible de proposer des implantations de l'IPL spécifiques à un environnement particulier qui permettent d'obtenir de meilleurs performances mais qui ne soient pas portables.

Le principal élément de l'IPL concerne la communication. Les communications avec

Ibis fonctionnent grâce à des ports d'émission et de réception. L'association de deux ports permet de définir un canal de communication unidirectionnel. Pour effectuer une communication, il est nécessaire de demander la création par le port d'émission d'un nouvel objet *message* et d'y insérer des éléments. Des types primitifs ainsi que des objets peuvent être ajoutés à un message. Il est également possible de demander l'envoi d'un intervalle de tableau. Lorsque le message est prêt, la primitive `send` permet d'envoyer le message construit. Un message peut aussi être transmis de façon asynchrone au fur et à mesure de sa construction, la méthode bloquante `finish()` permet alors de s'assurer que la transmission du message est effectivement terminée.

Deux possibilités s'offrent au développeur pour réceptionner un message : l'appel à la primitive bloquante `receive` du port de réception ou l'utilisation d'un mécanisme événementiel. Dans ce dernier cas, à l'arrivée d'un nouveau message, une méthode fournie par l'utilisateur sera appelée avec en paramètre le nouveau message reçu.

3.3 L'interface JToe

Comme nous l'avons vu précédemment, différentes API ont déjà été proposées pour permettre le remplacement, le plus facilement possible, du mécanisme de *serialization*. Cependant, cela aboutit souvent à une séparation artificielle entre le processus de *serialization* lui-même et la couche de communication. Par exemple KaRMI [100] définit la notion de *technologie*. La *serialization* d'un objet est effectuée par KaRMI et la *UKASerialization* puis le résultat produit est transmis à une technologie donnée. De nouvelles technologies peuvent être proposées pour améliorer les performances réseaux ou pour permettre le support de nouveaux types de réseaux. Cependant toutes les optimisations possibles dans le cadre du transfert d'objets ne peuvent s'appliquer à ce genre de structure. Les données à transmettre peuvent dépendre des machines virtuelles communicantes [14, 76] et/ou des mécanismes de communication disponibles. Le fait que les deux machines virtuelles communicantes soient identiques peut permettre de transmettre directement des zones mémoires. De même, il est possible d'exploiter des couches de communication permettant de ne pas *bufferiser* les données à transmettre. La séparation du mécanisme de *serialization* de celui du transport interdit la mise au point d'optimisations exploitant ces caractéristiques. Une telle séparation n'est donc pas souhaitable. Dans le cas de RMIX, au contraire, il est possible de proposer une implantation alternative globale pour l'appel de méthode à distance. Les mécanismes de *serialization* et de communication ne sont alors plus disjoints. Mais cela impose de gérer l'ensemble des problèmes liés à l'appel de méthode à distance : gestion des processus légers, des références distantes, ramasse-miettes distribué, etc. Enfin, Ibis se rapproche des objectifs de JToe avec l'IPL. JToe propose le transfert d'objets tandis que l'IPL celui de messages pouvant contenir des objets.

L'API JToe [28, 29, 73], basée sur les observations précédentes, est composée de deux interfaces : `Node` et `CopyListener` (Cf. programme 3.19). Pour recopier un objet sur

```
1  public interface Node {
2      void copy(Serializable object) throws JToeException;
3  }
1  public interface CopyListener {
2      void copied(Serializable object);
3  }
```

Programme 3.19: L'API JToe

un nœud distant, il est nécessaire d'obtenir l'objet `Node`[3] représentant le nœud cible puis d'invoquer sa méthode `copy` en passant l'objet à copier en argument. Du côté serveur, c'est la couche JToe qui est chargée de prévenir l'application de l'arrivée d'un nouvel objet grâce à un mécanisme événementiel, en appelant la méthode `copied` de l'interface `CopyListener`. C'est l'émetteur qui prend l'initiative de transmettre un objet. Aucun processus applicatif n'a besoin d'attendre pour qu'un objet puisse être reçu car c'est à l'implantation de JToe d'assurer la réception de tout nouvel objet puis de signaler cette arrivée à l'application.

Lors de la définition de l'API JToe, l'un des objectifs annoncé page 67 était qu'elle soit minimale afin de simplifier sa compréhension et son implantation. Le programme 3.19 permet de se convaincre que cet objectif est atteind puisque seules les primitives strictement nécessaires au transfert d'objets ont été retenues.

Un autre objectif était de n'imposer aucune limitation dans les possibilités d'implantation de cette API. En effet, nous avons vu précédemment que chaque système a sa propre approche pour améliorer les performances de la *serialization* : Ibis propose la génération de code pour la *serialization*, KaRMI demande au développeur d'écrire lui-même ce code, Expresso impose le principe de *clusters* d'objets. JToe, par contre, ne définit aucunement de quelle façon cette *serialization* doit se comporter ni comment elle interagit avec la couche de communication. Comme on peut le voir sur la figure 6 page suivante, aucune restriction n'existe sur la façon dont l'interface `Node` peut être implantée : la méthode `copy` est responsable de tout le processus de transfert d'un objet, c'est-à-dire le calcul des données à transférer – *serialization* – et le transfert effectif de ces données. Cela évite d'introduire une frontière artificielle entre ces deux opérations et permet d'envisager l'implantation de n'importe quel type d'optimisation du mécanisme de *serialization*.

Trois implantations de cette API ont été développées. Deux exploitent le mécanisme standard de *serialization* et utilisent, l'une RMI, l'autre TCP pour les communications. La troisième est spécifique à la machine virtuelle JikesRVM et exploite des mécanismes internes de cette dernière. Ces trois versions sont disponibles [73] sous licence *GNU Lesser General Public License* [51].

[3] JToe ne permet pas de retrouver un `Node` donné. Ce problème n'est pas lié au transfert d'objets mais au nommage. Un service de nommage généraliste quelconque peut-être utilisé.

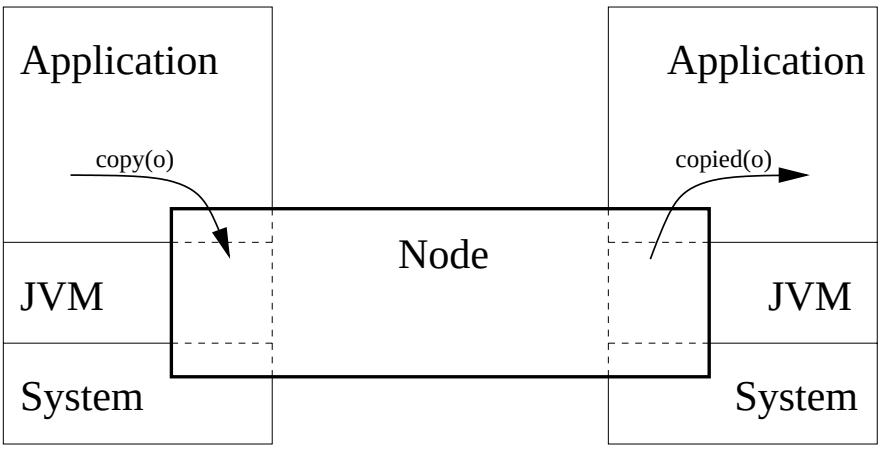


FIG. 6 – Fonctionnement général de JToe.

3.4 Implantation de JToe sur JikesRVM

Nous allons présenter en détail une implantation de JToe exploitant les spécificités de la machine virtuelle JikesRVM. Cette machine virtuelle met à disposition des mécanismes non standard qui sont exploités dans cette implantation de JToe pour fournir un outil le plus efficace possible. Nous allons commencer par un aperçu de cette machine virtuelle puis nous verrons comment nous l'exploitons pour accéder directement à la représentation en mémoire des objets ou pour structurer une zone mémoire brute en un ensemble d'objets. Nous étudierons également l'impact, sur notre code, des interactions poussées que nous entretenons avec le ramasse-miettes.

3.4.1 Un aperçu de JikesRVM

JikesRVM est une machine virtuelle Java développée par IBM et aujourd'hui disponible [71] sous licence *Common Public License* [70]. Le RVM de JikesRVM signifie *Research Virtual Machine*. Cette machine virtuelle est principalement écrite en Java mais ne nécessite pas de machine virtuelle Java pour s'exécuter. C'est en réalité un compilateur à la volée (*Just In Time compiler* en anglais). Ainsi, une machine virtuelle Java tiers n'est nécessaire qu'une seule fois, pour générer le véritable exécutable JikesRVM : une autre machine virtuelle exécute JikesRVM sur elle-même afin qu'elle se compile elle-même. Cela produit un exécutable spécifique à une architecture donnée, c'est JikesRVM.

JikesRVM étant principalement écrite en Java, des fonctionnalités non standard sont nécessaires afin de pouvoir effectuer certaines opérations bas niveau normalement impossibles en Java. Pour implanter un ramasse-miette, par exemple, il est nécessaire d'avoir accès à la mémoire et de pouvoir manipuler des adresses mémoire, ce qui est totalement interdit en Java. JikesRVM propose donc un certain nombre de classes – la plus importante étant `VM_Magic` – pour permettre cela. Lorsqu'une méthode d'une de ces classes

spéciales est invoquée, l'appel est intercepté par le compilateur à la volée qui génère le code machine approprié. Le fait que JikesRVM soit principalement écrite en Java ne signifie pas pour autant qu'elle soit directement portable puisqu'un compilateur à la volée est spécifique à une plate-forme cible. D'autre part, les classes qui utilisent `VM_Magic` ou une autre classe spéciale ne peuvent pas fonctionner avec une autre machine virtuelle que JikesRVM.

Nous présentons maintenant les fonctionnalités principales de JikesRVM. Cette machine virtuelle étant expérimentale, certains détails de la présentation qui suit peuvent être amenés à évoluer en même temps que la machine virtuelle. Cette description s'applique à la version 2.3 de JikesRVM.

Accès mémoire

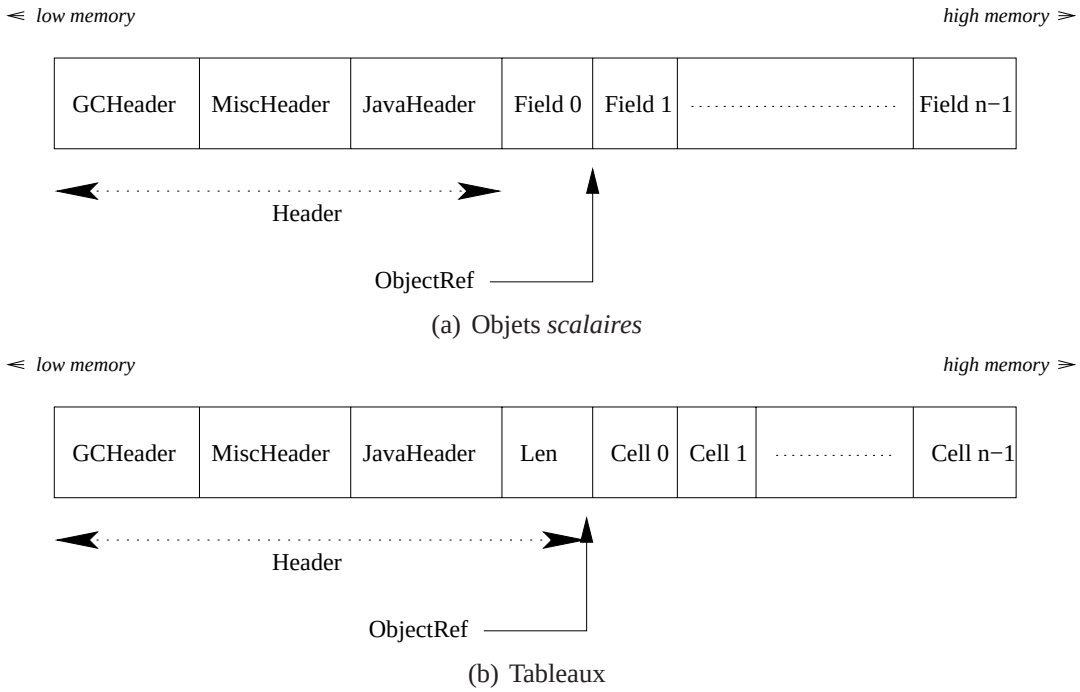
Un programme Java standard peut obtenir des informations sur un objet ou modifier la structure d'un objet grâce au mécanisme de réflexion. En plus de cette possibilité, JikesRVM permet un accès direct à la représentation bas niveau, en mémoire, d'un objet.

La méthode `VM_Magic.objectAsAddress()` retourne l'adresse mémoire effective d'un objet, représentée par *ObjectRef* sur la figure 7 page ci-contre. Un pointeur dans JikesRVM est représenté par une instance de la classe spéciale `VM_Address`. Cette classe représente une adresse mémoire et permet d'effectuer des opérations d'arithmétique des pointeurs ou de déréférencement pour obtenir la donnée stockée à une certaine adresse. Elle peut éventuellement être utilisée pour accéder à des objets ou les modifier directement au travers de leur représentation mémoire.

La classe `VM_ObjectModel` définit la représentation mémoire d'un objet Java. Les objets se découpent en deux catégories : objets scalaires (figure 7(a) page suivante) et tableaux (figure 7(b)). Ces deux types d'objets sont structurés en deux parties : la première contient l'en-tête de l'objet, c'est-à-dire un ensemble de données utilisées pour sa gestion par la machine virtuelle ; la seconde partie de l'objet contient les données effectives de cet objet.

La structure de l'en-tête est quasiment similaire pour un objet scalaire ou un tableau. Il est composé de trois parties dénommées `JavaHeader`, `MiscHeader` et `GCHeader`. `JavaHeader` et `GCHeader` sont utilisés pour la gestion des objets. Le type d'un objet, par exemple, est stocké dans `JavaHeader`. `MiscHeader` n'est pas utilisé par JikesRVM mais est disponible pour des utilisations particulières. Nous décrirons page 81 comment nous exploitons ce `MiscHeader` pour le calcul efficace d'un graphe d'objets. Par défaut, `MiscHeader` a une taille nulle mais cela peut être changé à la compilation de JikesRVM.

La partie *données* d'un objet contient les données effectives de l'objet, c'est-à-dire les valeurs de ses champs dans le cas d'un objet scalaire ou les valeurs de ses cellules dans le cas d'un tableau.



(a) Objets *scalaires*

(b) Tableaux

FIG. 7 – Représentation mémoire des objets

Interactions avec le ramasse-miettes

Un programme Java standard peut interagir avec le ramasse-miettes de différentes façons : il peut inciter ce ramasse-miettes à s'activer grâce à la méthode `System.gc()` ; il peut spécifier des opérations à effectuer avant la libération d'un objet en redéfinissant la méthode `finalize()` ; enfin, le paquetage `java.lang.ref` définit un certain nombre de mécanismes avancés pour interagir avec le ramasse-miettes.

JikesRVM propose des opérations plus intrusives. Un programme peut, par exemple, désactiver le ramasse-miettes pendant une période donnée – lorsqu'il manipule des pointeurs – ou allouer une zone mémoire *brute* qu'il pourra ensuite transformer en un véritable objet qui sera pris en charge par le ramasse-miettes.

Il est nécessaire, pour effectuer des opérations sur les pointeurs, de désactiver le ramasse-miettes – les données pouvant être déplacées par celui-ci – mais il s'agit d'une opération risquée. Il est, par exemple, totalement interdit d'allouer de la mémoire pendant que le ramasse-miettes est désactivé puisqu'une telle allocation mémoire pourrait nécessiter une intervention du ramasse-miettes pour libérer l'espace nécessaire. D'autre part, afin d'éviter que d'autres processus légers allouent de la mémoire, la désactivation du ramasse-miettes provoque également la désactivation de l'ordonnanceur. Les méthodes `VM.disableGC()` et `VM.enableGC()` permettent, respectivement, de désactiver et de réactiver le ramasse-miettes.

Il est également possible de découper le processus de création d'un objet en

deux phases : (1) allocation de la mémoire *brute* ; (2) initialisation de la zone mémoire comme un objet. Après cette initialisation, l'objet est pris en charge par le ramasse-miettes. Cette initialisation peut être effectuée, par exemple, grâce à la méthode `VM_ObjectModel.initializeScalar()`. L'allocation mémoire pour un objet peut se faire grâce à la méthode `Plan.alloc()`. Nous verrons également dans la section suivante qu'avec une connaissance spécifique du ramasse-miettes utilisé, il est possible de n'effectuer qu'une seule allocation mémoire pour un ensemble d'objets. La zone mémoire peut alors être manipulée et initialisée comme un bloc puis chacun des objets signalé au ramasse-miettes.

Malgré les différents risques liés à la désactivation du ramasse-miettes, cette désactivation est obligatoire pour certaines parties de notre implantation de JToe. Nous y reviendrons page 85.

3.4.2 Besoins supplémentaires pour JToe

L'implantation de JToe sur JikesRVM nécessite un certain nombre de mécanismes qui ne sont pas directement disponibles.

Communications zéro copie

Afin d'assurer un maximum de performances, nous souhaitons effectuer des communications sans copie intermédiaire. Cependant, l'envoi de données en Java s'effectue toujours au travers de tableaux d'octets. Les données à transmettre doivent donc systématiquement être converties et copiées dans des tableaux d'octets. Pour limiter les recopies, nous avons besoin d'un outil de communication qui puisse directement envoyer des octets à partir d'un pointeur (une `VM_Adress` avec JikesRVM) et recevoir des données directement vers une zone mémoire spécifiée par un pointeur.

Puisque JikesRVM permet d'obtenir l'adresse mémoire effective d'un objet Java, il est donc possible d'effectuer les transferts d'objets en zéro copie. A cet effet, nous avons développé la classe `ZSocket` que nous présenterons page 82. Celle-ci repose sur des primitives bas-niveau de JikesRVM afin de transmettre directement la représentation mémoire d'un objet Java.

Allocation d'un ensemble d'objets

Comme nous le verrons lorsque nous décrirons l'implantation de JToe, nous souhaitons pouvoir stocker un ensemble d'objets dans une zone mémoire contiguë, pour des raisons de communication et pour limiter le nombre d'allocations mémoire. Pour cela, nous allouons une zone mémoire *brute*, dont la taille correspond à la somme des tailles nécessaires pour le stockage de chacun des objets. Une fois cette zone mémoire initialisée, nous convertissons chacun de ces objets en un objet Java normal grâce à la méthode

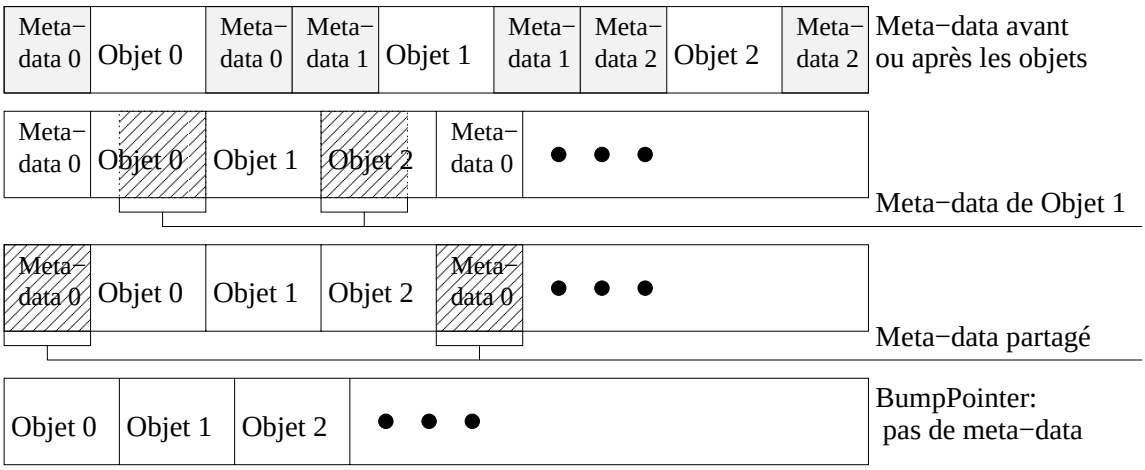


FIG. 8 – Informations stockées par le ramasse-miettes

`VM_Object.initializeScalar()` que nous avons déjà évoquée. La question qui se pose ici est de savoir comment allouer cette zone mémoire ou, plus exactement, où l'allouer.

Lorsqu'on alloue de la mémoire dans JikesRVM, la supposition principale qui est faite est que cette zone mémoire ne sera dédiée qu'à un seul objet Java. Le ramasse-miettes peut alors stocker des informations sur cet objet avant ou après la zone mémoire qu'il lui a allouée. Cela nous interdit d'effectuer une seul allocation pour y stocker plusieurs objets comme illustré figure 8. Sur cette figure, les parties notées *Objet* i correspondent à une zone mémoire contenant l'objet i, y compris les en-têtes de cet objet. Les zones notées *Meta-data* i correspondent à des données du ramasse-miettes qui concernent l'objet i. Ces données s'ajoutent à celles déjà présentes dans `GCHeader`.

On constate sur les deuxième et troisième schémas de la figure 8 que stocker plusieurs objets dans une même zone mémoire contiguë peut provoquer une corruption des données des objets par le ramasse-miettes qui supposera que la zone avant ou après chaque objet est libre pour ses propres données, ou une incohérence des données du ramasse-miettes qui seront partagées pour l'ensemble des objets.

Différents type d'*espaces* sont gérés par le ramasse-miettes. Les objets sont alloués dans, et éventuellement déplacés entre, différents espaces en fonction de leur taille, leur âge ou tout autre caractéristique pertinente pour le ramasse-miettes. Il existe un espace particulier – implanté par la classe `BumpPointer` – dans lequel aucune donnée n'est ajoutée par le ramasse-miettes avant ou après la zone mémoire d'un objet. Cet espace est utilisé par différents ramasse-miettes[4], principalement comme nurserie. Afin de pouvoir effectuer une seule allocation mémoire pour un ensemble d'objets, il nous faut faire cette allocation explicitement dans cet espace. Afin d'accéder à l'espace `BumpPointer` d'un ramasse-miettes donné, il nous faut écrire un code spécifique au ramasse-miettes cible. Pour cela, nous avons créé la classe `GcManipulator` que nous présenterons page 86.

[4]Lors de la compilation de JikesRVM, il est possible de choisir entre différents types de ramasse-miettes.

Gestion des *gros* objets

Les ramasse-miettes proposés par JikesRVM gèrent les *gros* objets de façon spécifique et les allouent dans un espace spécifique. Puisque les ramasse-miettes gèrent les gros objets de façon spécifique, il ne serait pas cohérent de les mélanger avec les *petits* objets. Afin de prendre ces objets en compte correctement, nous devons connaître le seuil à partir duquel un objet est considéré comme *gros*. C'est une autre des fonctionnalités proposées par notre classe `GcManipulator`.

3.4.3 Implantation de JToe sur JikesRVM

Pour réaliser JToe, il est nécessaire d'implanter la méthode `copy()` de l'interface `Node` ainsi que le support permettant d'appeler la méthode `copied()` de l'interface `CopyListener`.

L'objectif de notre implantation de JToe sur JikesRVM est d'être le plus efficace possible dans le cas de deux machines similaires exécutant la même machine virtuelle Java. Ces deux machines virtuelles étant identiques et s'exécutant sur les mêmes plate-formes matérielles et logicielles, nous considérons qu'elles partagent le même ordre des octets (*byte ordering*) ainsi que la même représentation mémoire des objets et le même type de ramasse-miettes.

A partir de ces hypothèses, le transfert d'objets se déroule en trois phases, comme présenté figure 9 page suivante :

(1) Calcul de l'ensemble des objets à transférer.

L'état d'un objet est défini par l'état des objets qu'il référence. Lors de l'envoi d'un objet, il est donc nécessaire de trouver les objets que celui-ci référence. Nous décrirons cette étape page ci-contre.

(2) Envoi des données *brutes* des objets en zéro copie.

La représentation mémoire bas niveau des objets est transférée directement, sans copie intermédiaire. Côté récepteur, une zone mémoire de taille suffisante est allouée au début de la communication et les données sont lues directement vers cette zone mémoire, là aussi sans copie intermédiaire. Cette étape est décrite page 82.

(3) Restauration du graphe.

Des données supplémentaires sont transmises. Elles permettront de reconstruire le graphe d'objets sur le récepteur. Lorsque toutes les références entre objets ont été mises à jour, ces objets sont signalés au ramasse-miettes et seront, par la suite, considérés comme des objets à part entière.

Ce processus implique des interactions poussées avec le ramasse-miettes que nous décrirons page 85.

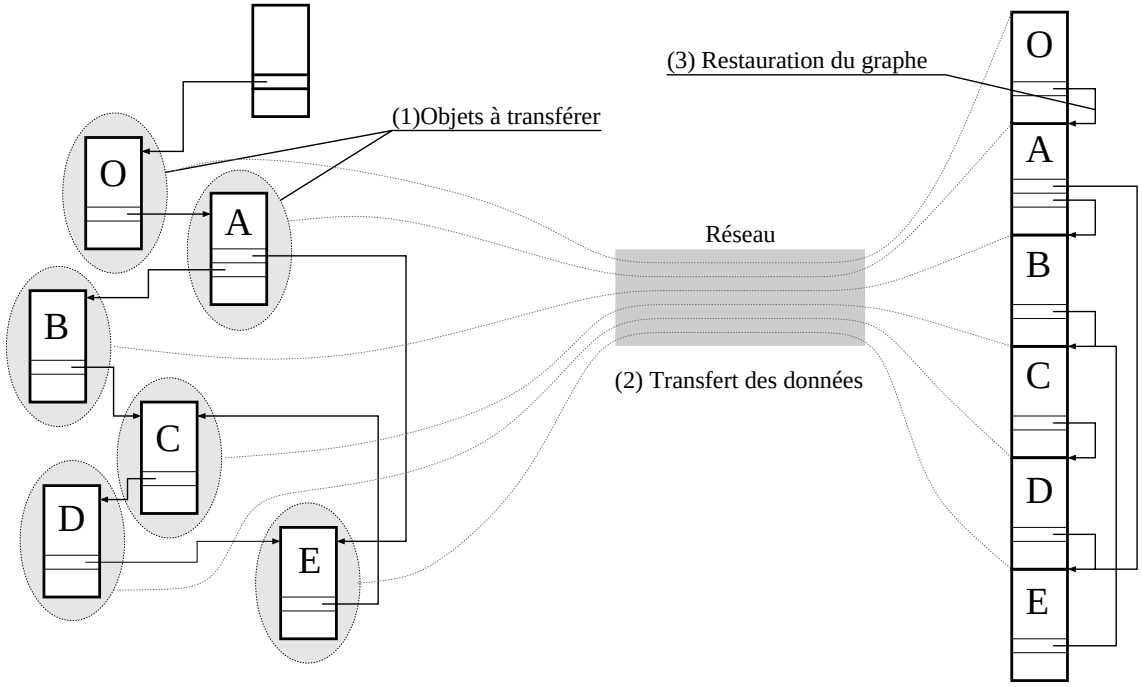


FIG. 9 – Transfert d'un objet

Calcul du graphe d'objets

Cette implantation de JToe est conçue pour remplacer le mécanisme de *serialization* standard de Java dans le contexte du calcul haute performance. Il se conforme donc à certains principes de ce mécanisme de *serialization*, en particulier l'envoi d'un objet implique une copie en profondeur du graphe d'objets dont il est la racine. La cohérence de ce graphe n'est pas garantie en cas de modification concurrente par une autre partie du programme pendant ce parcours et, de façon générale, pendant l'ensemble du mécanisme de transfert.

L'état d'un objet est défini, entre autres, par l'état des objets qu'il référence. La première chose à faire, avant de transmettre un objet, est donc de trouver l'ensemble des objets qui définissent son état et qui doivent être transmis. Pour faire cela, il est nécessaire de parcourir le graphe enraciné dont les nœuds sont des objets Java, la racine est l'objet que l'on souhaite transférer et les arêtes sont les références entre ces objets.

La figure 9 montre l'ensemble des objets à transmettre lorsqu'on souhaite envoyer l'objet $\circ$. Cet ensemble est calculé en parcourant le graphe des objets *utilisés* par $\circ$. L'algorithme, en lui-même, est trivial : pour tout objet référencé par $\circ$, s'il n'a pas déjà été visité, le marquer comme tel et l'ajouter à l'ensemble des objets à transmettre et calculer récursivement l'ensemble des objets à transmettre avec ce dernier.

En Java, il n'est pas possible d'ajouter un champ à un objet si cela n'était pas prévu au départ dans sa classe. Il n'est donc pas possible de marquer directement un objet comme

visité pour le parcours du graphe. Il est alors nécessaire d'utiliser une table dans laquelle tous les objets visités sont insérés, une recherche étant effectuée dans cette table lorsqu'un nouvel objet est rencontré. Ce mécanisme de recherche engendre un coût important. Pour éviter cela, nous exploitons le `MiscHeader` présenté page 76. Ainsi, nous ajoutons un champ au `MiscHeader` dont la valeur par défaut est 0. Nous plaçons cette valeur à 1 pour un objet visité. Dès la fin du parcours du graphe et avant toute transmission de données, la marque de chacun des objets visités est remise à 0.

Cette fonctionnalité nous a permis d'améliorer significativement les performances de JToe par rapport à l'utilisation d'une table pour identifier les objets visités. Cependant, cette solution nous empêche d'effectuer plusieurs parcours de graphe simultanés. En effet, si les parcours de deux graphes non disjoints sont en cours simultanément, il y aura conflit sur la marque pour les objets communs aux deux graphes, ce qui engendrera des erreurs dans le calcul de ces graphes. Une option serait d'utiliser un bit distinct du `MiscHeader` pour des parcours distincts du graphe. Cela autoriserait un nombre prédéfini de parcours de graphe à s'exécuter simultanément. Dans la version actuelle, un seul parcours à la fois est autorisé.

Communications en zéro copie

Nous avons développé la classe `ZSocket` qui fournit des opérations d'entrée/sortie en zéro copie.[5] Cette classe permet l'échange de données contenues dans des tableaux d'octets, que nous utilisons pour transmettre les données propres à notre protocole ; mais elle permet surtout d'envoyer une zone mémoire brute spécifiée par son adresse et sa taille. Le programme 3.20 page suivante présente l'API de cette classe.

La construction d'un objet `ZSocket` se fait à partir d'un objet `Socket` existant. Les classes `ZOutputStream` et `ZInputStream` permettent d'effectuer les opérations de communication. Les opérations sont répercutées sur le socket système grâce à la classe `VM_SysCall` proposée par JikesRVM et qui permet d'effectuer des appels systèmes natifs.

Les méthodes d'envoi ou de réception de données brutes reçoivent en argument un `VM_Address`, c'est à dire un pointeur, pour envoyer les données depuis ou recevoir des données vers cette adresse. La manipulation de pointeur doit se faire avec le ramasse-miettes désactivé. Typiquement, un programme appelant `writeZ()` commencera par désactiver le ramasse-miettes avant de calculer l'adresse depuis laquelle il souhaite envoyer les données – en utilisant, par exemple, `VM_Magic.objectAsAddress()` – puis appellera la méthode `writeZ()` et, enfin, réactivera le ramasse-miettes.

Le ramasse-miettes étant désactivé, aucune allocation mémoire ne peut être effectuée par les méthodes `writeZ()` ou `readZ()`. De plus, l'ordonnanceur étant désactivé en même temps que le ramasse-miettes, aucun changement de contexte n'est possible

[5] Nous parlons ici de *zéro copie* pour indiquer que notre code n'effectue aucune copie intermédiaire. Ce n'est pas forcément le cas de la couche de communication. Cette version de JToe utilise TCP.

```
1  public class ZSocket{
2    // Build a ZSocket from an initialized and connected socket
3    public ZSocket(Socket s);
4
5    // Retrieve a ZInputStream for read operations on the ZSocket
6    public ZInputStream getZInputStream() throws IOException;
7
8    // Retrieve a ZOutputStream for write operations on the ZSocket
9    public ZOutputStream getZOutputStream() throws IOException;
10 }
1  public abstract class ZOutputStream extends OutputStream{
2    // write regular byte array data.
3    public abstract int writeZ(byte b,
4                               int off,
5                               int len) throws IOException;
6    // write count bytes contained in memory starting at from address.
7    public abstract int writeZ(VM_Address from,
8                               int count) throws IOException;
9  }
1  public abstract class ZInputStream extends InputStream{
2    // read regular data into byte array.
3    public abstract int readZ(byte b,
4                              int off,
5                              int len) throws IOException;
6    // read count bytes from the socket and
7    // write them into memory, starting at dest.
8    public abstract int readZ(VM_Address dest,
9                              int count) throws IOException;
10 }
```

Programme 3.20: Les classes `ZSocket` et `Z(Input/Output)Stream`.

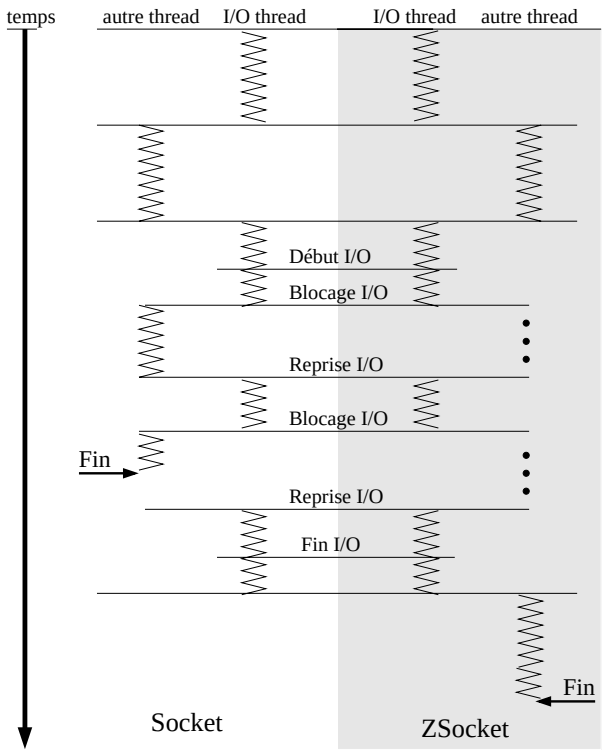


FIG. 10 – Impact du non recouvrement des communications avec un `ZSocket` par rapport à un `Socket` normal

pendant ces communications et en particulier, dans le cas d'une communication bloquante, il est nécessaire d'effectuer une attente active. Comme le montre la figure 10, cette contrainte interdit le recouvrement de la communication par un autre processus léger.[6]

Bien que cette absence de recouvrement ne soit pas perceptible dans les performances de JToe évaluées grâce à des programmes de type ping-pong, cela peut être un handicap pour des applications réelles. Une solution envisageable est de modifier la méthode `writeZ()` afin qu'elle reçoive, en lieu et place d'un pointeur, un objet Java standard et un décalage. L'adresse effective depuis laquelle le transfert doit s'effectuer devient alors une fonction de l'adresse de l'objet (récupérable grâce à `VM_Magic.objectAsAddress()`) et du décalage. L'attente active en cas de communication bloquante n'est plus nécessaire puisqu'il est toujours possible de retrouver l'adresse à partir de laquelle l'envoi doit reprendre grâce à ces informations. Cette solution nécessite cependant certaines modifications dans notre format de données ; nous reviendrons sur ce point section 3.5 page 89.

[6] Pour des raisons de place, le terme *thread* est utilisé figure 10 en lieu et place du terme *processus léger*.

Interactions avec le ramasse-miettes

JikesRVM propose des fonctionnalités permettant des interactions de bas-niveau avec le ramasse-miettes. Nous allons présenter nos besoins sur ce point dans le cadre de JToe et les contraintes que ces interactions imposent à notre implantation.

Nous souhaitons effectuer des transferts de données sans copie intermédiaire. Comme nous l'avons vu page 82, cela implique d'effectuer nos transferts en désactivant le ramasse-miettes puisque nous manipulons directement des pointeurs. Pendant ces périodes, le changement de contexte entre processus légers devient impossible, ce qui nous oblige à respecter un modèle de communication symétrique : lorsque nous transmettons des données, nous devons nous assurer que le destinataire est bien en attente de ces données ; réciproquement, lorsque nous nous mettons en attente de réception de données, nous devons nous assurer que les données sont en cours de transmission par l'émetteur.

Comme nous le verrons par la suite, lors de la reconstruction d'un graphe d'objets, nous utilisons les adresses originales des objets. Il est donc impératif que le ramasse-miettes ne les déplace pas pendant que nous les transmettons. Du coté du receveur, le ramasse-miettes doit également être désactivé lors de l'initialisation des références entre objets puisque, là encore, nous manipulons directement des pointeurs. Etant donné qu'il est impossible d'allouer de la mémoire au cours de ces deux opérations, notre code est conçu de façon à ce que toute la mémoire qui leur est nécessaire puisse être allouée *avant* que le ramasse-miettes ne soit désactivé. Ceci a une implication forte sur le format des données que nous transmettons : le nombre et le type respectif des objets que nous nous apprêtons à transmettre sont envoyés en premier. Cela permet au destinataire d'allouer toute la mémoire nécessaire avant de recevoir effectivement les objets. D'autre part, nous avons dû implanter une table d'association n'ayant pas recours à l'allocation mémoire afin de pouvoir reconstituer le graphe d'objets chez le destinataire.

L'allocation d'une seule zone mémoire pour stocker plusieurs objets reçus complique encore les interactions avec le ramasse-miettes. Comme nous l'avons évoqué page 80, les *gros* objets sont gérés spécifiquement par le ramasse-miettes. N'effectuer qu'une seule allocation en présence de gros objets aurait pour conséquence de les *mélanger* avec les *petits* objets. Nous souhaitons éviter ce mélange afin de nous rapprocher de la politique du ramasse-miettes. Cela nous oblige à être capable de déterminer si un objet est *petit* ou *gros*. Nous avons développé la classe `GcManipulator` (Cf. programme 3.21) qui nous fournit cette information grâce à sa méthode `isSmallObject()`. L'implantation de cette méthode dépend directement du ramasse-miettes utilisé. Cela nous permet de séparer les *gros* objets des *petits*, ces derniers étant stockés dans une seule zone mémoire contiguë.

La seule façon de stocker tous nos *petits* objets dans une seule zone mémoire contiguë tout en restant cohérent avec le ramasse-miettes est de faire cela dans l'espace `BumpPointer` que nous avons présenté page 78. La méthode

```
1 public class GcManipulator {
2   public boolean isSmallObject(Object o);
3   public VM_Address allocate(int bytes);
4 }
```

Programme 3.21: La classe GcManipulator

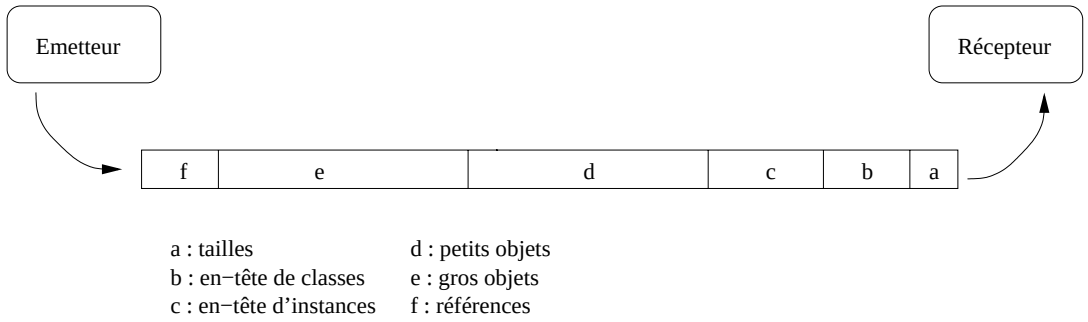


FIG. 11 – Format de données

`GcManipulator.allocate()` nous assure que cette allocation est effectivement effectuée dans le `BumpPointer` du ramasse-miettes. Cela implique que notre implantation spécifique de JToe sur JikesRVM ne peut fonctionner qu'avec des ramasse-miettes qui possèdent un espace de type `BumpPointer`. D'autre part, pour chaque type de ramasse-miettes avec lequel nous souhaitons fonctionner, il est nécessaire de développer une version distincte, spécifique à ce ramasse-miettes, de la classe `GcManipulator`.

Format des données

Nous allons maintenant décrire le format des données transférées par notre implantation de JToe. La figure 11 présente le flux des données pour le transfert d'un graphe d'objets. Ce flux est divisé en six parties :

(a) tailles :
contient six entiers qui représentent la taille de b+c, la taille de b, le nombre de noms de classes dans c, le nombre d'objets de type `java.lang.Class`, le nombre de *petits* objets et le nombre de *gros* objets ;

(b) en-tête de classes :
contient le nom de chaque classe utilisée dans le graphe d'objets. Le nombre de caractères de ce nom est indiqué avant chacun des noms de classe. La position du nom d'une classe devient l'identifiant de cette classe dans la suite. Cet identifiant permet de spécifier le type de chaque objet dans la suite du flux ;

(c) en-tête d'instances :
cet en-tête contient des informations sur chacun des objets transmis. Le premier élément décrit son genre : tableau, objet scalaire ou instance de la classe

`java.lang.Class`. Dans ce dernier cas, l'en-tête contiendra uniquement l'identifiant de la classe correspondante, cela suffira à reconstituer l'objet à l'arrivée. Dans le cas d'un tableau, le nombre de cellules est également ajouté. Pour un objet scalaire ou un tableau, l'en tête contient aussi la taille de cet objet ;

(d) *petits* objets :
cette partie du flux contient toutes les données brutes de tous les *petits* objets transmis. L'ensemble de la zone mémoire bas-niveau allouée à chaque objet – tel que présenté figure 7 page 77 – est transmis, y compris les en-têtes de JikesRVM. Toutes les données des *petits* objets peuvent alors être lues d'un bloc, sans avoir à se soucier des espaces nécessaires pour les en-têtes ;

(e) *gros* objets :
les données de chacun des gros objets transmis sont contenues dans cette partie du flux. Seule la partie données des objets – comme décrite figure 7 page 77 – est transmise. Afin d'être cohérent avec la politique de gestion des *gros* objets par le ramasse-miettes, le destinataire alloue et initialise chacun des gros objets séparément ;

(f) références :
cette partie du flux contient les adresses mémoires, sur l'émetteur, de chacun des objets transmis. Celles-ci sont envoyées dans le même ordre que les objets l'ont été. Elles permettent au destinataire de reconstituer les références du graphe d'objets.

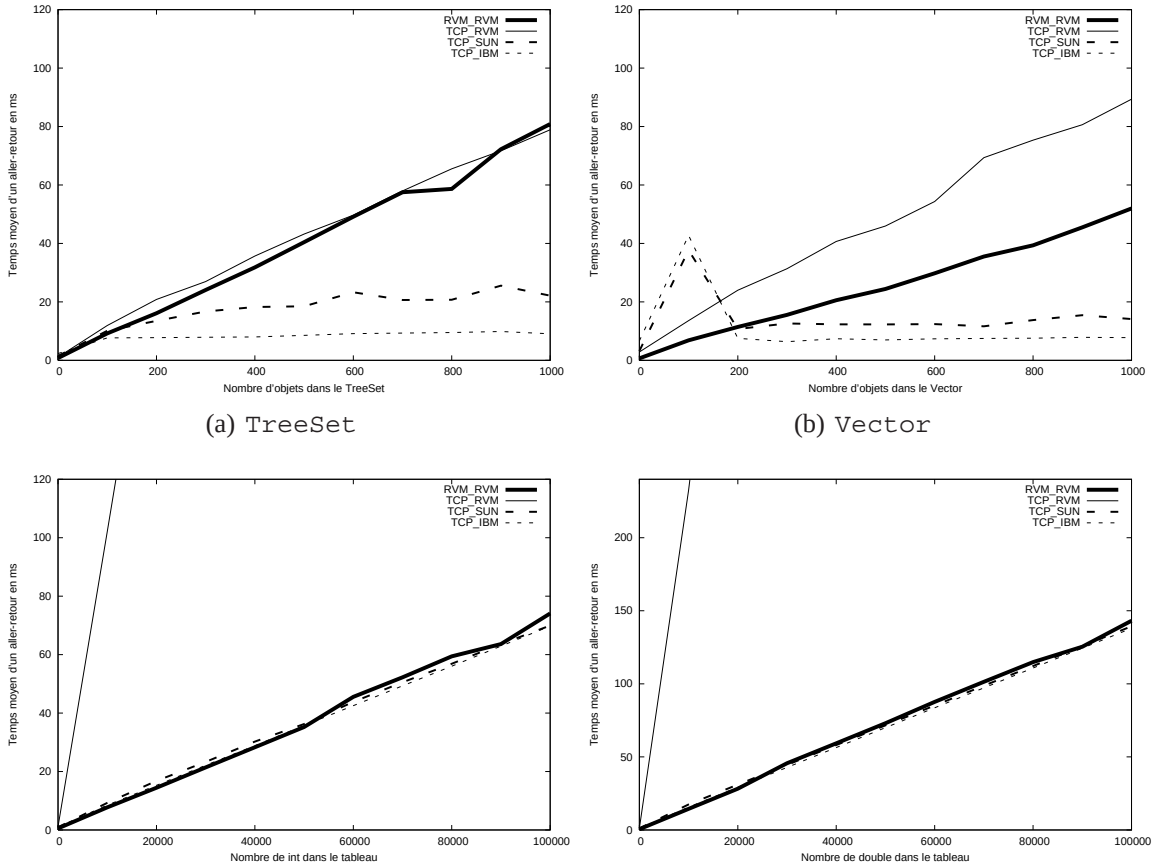
Les principales informations du protocole – a, b, et c – sont transmises avant les véritables données des objets. Cela nous permet de nous assurer que toutes les classes nécessaires ont été chargées dans la machine virtuelle et que toute la mémoire nécessaire a été allouée avant de désactiver le ramasse-miettes pour la réception des données des objets. Le ramasse-miettes ne sera réactivé qu'une fois toutes les données reçues et les références entre les objets mises à jour.

Comme nous l'évoquions page 85, les *petits* objets sont stockés ensemble dans une même zone mémoire. Nous les envoyons donc ensemble afin de permettre de recevoir leur données d'un bloc.

3.4.4 Tests de performance

La figure 12 page suivante présente les temps de transfert d'objets avec JToe spécifique à JikesRVM et JToe 100% Java. La version 100% Java exploite le mécanisme standard de la *serialization*. Dans les deux cas, les communications s'effectuent sur TCP. Les courbes correspondent au temps moyen d'un aller-retour pour un objet donné dans une application de type ping-pong.

– La courbe `RVM_RVM` correspond à la version de JToe spécifique à JikesRVM s'exécutant sur JikesRVM version 2.3 *FastAdaptiveSemiSpace* – on notera que ce code ne peut pas s'exécuter avec une autre machine virtuelle que JikesRVM.



(a) `TreeSet` (b) `Vector`

(c) Tableau de `int` (d) Tableau de `double`

FIG. 12 – Tests JToe : temps moyen d'un aller-retour pour un objet donné avec JToe.

- La courbe `TCP_RVM` correspond à l'implantation 100% Java de JToe s'exécutant sur la même version de JikesRVM.
- Les courbes `TCP_SUN` et `TCP_IBM` correspondent à l'implantation 100% Java de JToe s'exécutant, respectivement, sur la machine virtuelle Java de Sun version 1.4.1_01-b01 et la machine virtuelle Java d'IBM version 1.4.1.

Ces tests ont été réalisés sur deux PC équipés d'un processeur Intel 1.7GHz, de 512MB de mémoire vive, utilisant le système d'exploitation Linux version 2.4.20 et connectés par un réseau Ethernet non dédié de 100Mbps.

De façon générale, l'exécution de notre application de ping-pong sur cette version de JikesRVM est plus lente que sur les deux autres machines virtuelles.

Dans le cas de l'échange d'objets de type `Vector`, on constate que la version spécifique de JToe est 40% à 50% plus rapide que la version 100% Java lorsque les deux s'exécutent sur JikesRVM.

En ce qui concerne les objets de type `TreeSet`, la version spécifique de JToe n'est pas plus performante que celle ayant recours au mécanisme de *serialization* standard. Cela

peut s'expliquer par le fait que la classe `java.util.TreeSet` exploite le mécanisme standard de *serialization* pour proposer une méthode d'encodage ad-hoc qui envoie directement vers le flux de *serialization*, et de façon linéaire, les objets référencés. La version spécifique de JToe ne supportant pas la *serialization* ad-hoc, elle doit donc explorer totalement le graphe d'objets. Une autre raison de cette contre-performance vient du fait que, comme nous l'avons évoqué section 3.4.3 page 82, l'implantation actuelle ne permet pas le recouvrement des communications et surtout, que le format de données impose à l'émetteur de connaître l'ensemble de tous les objets à transmettre avant de pouvoir commencer la transmission, ce qui empêche de paralléliser la réception de données avec le calcul des prochaines données à transmettre. Dans le cas de la version 100% Java de JToe, ce problème n'existe pas et l'application peut donc bénéficier d'un effet *pipeline*.

Enfin, dans le cas des tableaux d'entiers ou de doubles, la version spécifique de JToe est nettement plus performante que celle utilisant la *serialization* standard sur JikesRVM. De plus, les temps obtenus sont comparables à ceux observés sur les machines virtuelles Sun ou IBM malgré le handicap de cette version de JikesRVM en terme de performance.

3.5 Perspectives

Premièrement, comme évoqué page 82, il nous faut supprimer l'attente active en émission de notre couche de communication. Pour cela, il nous est nécessaire de modifier à la fois la couche de communication elle-même et notre format de données défini page 86. Cela permettra d'envisager le recouvrement des communications en émission.

Deuxièmement, nous pensons également que la version spécifique de JToe peut bénéficier d'un effet *pipeline* et que cela peut aider à résoudre certains problèmes de performance, en particulier ceux observés lors du transfert de `TreeSets` page 87. Cela implique une modification en profondeur de notre format de données afin de pouvoir commencer à transmettre des données avant que l'ensemble de tous les objets à transmettre ne soit connu. Puisque les données ne seront plus reçues d'un bloc, il est indispensable de supprimer l'attente active côté récepteur. Cette suppression signifie que le ramasse-miettes sera actif entre plusieurs phases de réception d'un même graphe d'objets. La difficulté consiste à gérer le fait que, parmi les objets déjà reçus, certains peuvent référencer des objets non encore reçus, ce qui pose un problème pour le fonctionnement du ramasse-miettes.

Troisièmement, l'un des problèmes majeurs de notre travail est la compatibilité avec le mécanisme standard de *serialization*. Nous ne souhaitons pas être compatible au sens du protocole ou du format de données mais nous souhaitons permettre la *serialization* ad-hoc. En Java, une classe peut définir la façon dont ses instances seront *serializées*. Cela peut être extrêmement utile, non seulement pour des raisons de performance mais surtout pour des raisons de cohérence. Par exemple, un objet utilisant une connexion réseau ne va pas transmettre celle-ci telle quelle, ce qui n'aurait pas de sens, mais transmettra éventuellement les informations nécessaires pour la rétablir. En l'état actuel de l'implantation

de JToe sur JikesRVM, la cohérence des objets ayant recours à ce type de *serialization* ad-hoc n'est pas garantie.

L'implantation de JToe sur JikesRVM exploite des mécanismes de bas niveau afin de permettre la communication, sans copie intermédiaire, d'objets Java. L'exploitation réelle des possibilités que cela offre nécessiterait l'utilisation de couches de communication plus performantes que TCP et permettant effectivement la communication en zéro copie [96, 115, 138]. D'autre part, et comme on peut le percevoir tout au long de la section 3.4, cette interaction de bas niveau a un prix. On se rend compte que le gain apporté par l'accès direct à la mémoire est diminué par le coût qu'il engendre en terme de complexité du code de JToe, mais surtout en terme de performances : la désactivation du ramasse-miettes et de l'ordonnanceur interdit le recouvrement des communications et impose d'avoir toute l'information sur le graphe à transmettre avant que la moindre transmission soit possible. Il est donc difficile de prédire l'impact des optimisations mises en œuvres si on souhaitait les appliquer à d'autres machines virtuelles comme celles de Sun ou d'IBM : les performances qu'elles présentent sur les différentes courbes de la figure 12 page 88 seraient-elles améliorées ou, au contraire, dégradées ?

Pour revenir à l'API JToe elle-même, on peut noter que celle-ci est, volontairement, sous-spécifiée. Rien ne précise si la *serialization* ad-hoc doit être supportée ou non, par exemple. Cette sous-spécification évite de fixer des limites déraisonnables dès le début mais implique également des évolutions pour préciser un certain nombre de points. En particulier, en ce qui concerne la synchronisation de la méthode `copy()` rien, aujourd'hui, ne précise si la transmission est terminée à la fin de l'appel (et les données transmises réutilisables par l'appelant) ou si la transmission se fait de façon asynchrone (dans ce cas l'objet transmis ne doit pas être modifié avant la fin effective de la transmission). Toutes les implantations actuelles de JToe sont synchrones mais des versions asynchrones présenteraient également un intérêt, en particulier pour bénéficier d'un effet *pipeline* lors de la transmission de plusieurs objets à la suite. C'est pourquoi il nous paraît nécessaire de faire évoluer l'API JToe pour faire apparaître l'aspect potentiellement asynchrone de cette méthode `copy()`, par exemple en ajoutant une valeur de retour qui permette, par la suite, de s'assurer que l'opération est bien terminée, à l'image de la méthode `finish()` de Ibis que nous évoquions page 73. On se rapproche alors des mécanismes d'appel de méthode asynchrone sur lesquels nous reviendrons au chapitre 4.

Deuxième partie

Prise en charge de la latence

Chapitre 4

Latence : appel asynchrone de méthode à distance

Nous présenterons au chapitre suivant nos travaux sur l'activabilité dont l'objectif est la transformation automatique d'appels de méthode synchrones en appels asynchrones afin, dans le cas distant, de recouvrir ces appels comme nous l'avons vu section 1.5.2 page 42. Avant cela, il nous semble pertinent de présenter les principes des appels de méthode asynchrones. Nous allons étudier ces mécanismes au travers des fonctionnalités proposées par trois plates-formes : ProActive [20], Mandala [135] et E [92].

L'appel de routine à distance permet l'interaction entre deux machines, ou plus généralement deux processus distincts, le premier demandant l'exécution d'une commande au second. Nous avons vu différentes possibilités dans ce domaine dans la section 1.4.2 page 30. Lorsque cet appel distant est synchrone, l'appelant est bloqué et ne peut continuer son exécution tant que le résultat de sa commande ne lui est pas revenu. Ce temps d'attente inclut le délai nécessaire au transit de la commande sur le réseau, à son exécution puis au transit du résultat. Tout ce temps pourrait parfois être recouvert et exploité. L'appel asynchrone permet cela : on appelle une routine avant d'avoir besoin de son résultat ; pendant que la routine s'exécute en parallèle, on continue son exécution.

4.1 Principe de l'appel de méthode asynchrone

Le principe de l'appel de méthode asynchrone est de découpler l'invocation des méthodes sur un objet de leur exécution. L'appel asynchrone d'une méthode retournera immédiatement un résultat, appelé *futur*,[1] qui permettra de consulter le véritable résultat produit par l'exécution de cette méthode. Cette exécution s'effectue de façon asynchrone dans un flot d'exécution spécifique.

[1]Nous emploierons le terme *futur* qui est celui utilisé pour ProActive et Mandala. E utilise le nom *promise*.

Différents modèles d'asynchronisme existent. ProActive repose sur les objets actifs tels que décrits par Caromel [17]. Dans ce cas, un objet actif est associé à *un et un seul flot d'exécution* chargé d'exécuter les méthodes invoquées sur l'objet. Dans ce modèle, tous les objets ne sont pas actifs. Les objets non actifs sont dits *passifs*. Le partage d'un même objet passif entre plusieurs objets actifs est interdit. Le passage des paramètres par copie entre objets actifs garantit cela. Une routine est chargée de parcourir la liste des appels effectués sur l'objet et de les exécuter. Le comportement par défaut de cette routine consiste à exécuter séquentiellement les appels de méthode en attente, dans l'ordre dans lequel les appels ont été reçus mais cela peut être redéfini.[2]

Mandala repose sur le principe des références asynchrones [135]. Puisque l'asynchronisme se décide au niveau de la référence sur un objet, plusieurs références différentes et ayant chacune leur propre politique de gestion de l'asynchronisme peuvent partager un même objet. Par ailleurs, les références peuvent être chaînées entre elles, ce qui permet d'implanter, si on le souhaite, un asynchronisme coté client *et* coté serveur : une première référence asynchrone côté client est chaînée avec une seconde côté serveur. Enfin, plusieurs processus légers peuvent accéder simultanément à un objet si la politique de la référence pointant directement sur cet objet le permet ou par un seul si la politique l'exige. Mandala propose différents moyens d'expression de l'asynchronisme. Toutes nos remarques dans la suite de ce chapitre s'appliquent à la *semi-transparence* [135, p. 186, chap. 16] même dans le cas où nous ne le précisons pas.

La plate-forme E propose la notion de *vat*. Un *vat* est un espace mémoire virtuel contenant un ensemble d'objets associés à un processus. Toutes les méthodes invoquées sur un objet du *vat* sont exécutées par ce processus. Les appels de méthode reçus par le *vat* sont exécutés séquentiellement par le processus associé, dans l'ordre de leur arrivée. Plusieurs objets d'un même *vat* peuvent être accessibles à distance. Leurs méthodes seront exécutées par l'unique processus du *vat*, séquentiellement.

Ces trois plates-formes nous permettent d'étudier trois modèles d'asynchronisme différents. Nous allons voir les différentes stratégies qui sont mises en œuvre dans chacune d'entre elles pour synchroniser appelant et appelé ainsi que pour gérer les exceptions dans un cadre asynchrone. Nous étudierons ensuite les différents niveaux de masquage de l'asynchronisme par ces plates-formes.

4.2 Synchronisation

L'appel asynchrone de méthode nécessite la mise en place de différents points de synchronisation entre appelant et appelé pour transmettre l'appel et ses arguments à l'appelé puis pour retourner le résultat à l'appelant. Nous allons voir l'utilisation du rendez-vous

[2] Cela permet de gérer les contraintes de gestion de structures de données bornées. Si on considère un *buffer* borné, alors tant que le *buffer* est plein, la routine n'accepte que l'exécution de méthodes retirant des éléments du *buffer* ou, au moins, n'en ajoutant pas.

pour la transmission d'un appel à distance ainsi que de l'attente par nécessité et l'utilisation anticipée du résultat d'un appel.

4.2.1 Rendez-vous

La plate-forme ProActive a recours au mécanisme de rendez-vous. Il n'est pas présent dans E et peut être utilisé ou non avec Mandala. Lors d'un appel asynchrone avec rendez-vous, l'appelant reste bloqué le temps de ce rendez-vous, c'est-à-dire le temps que l'appel de méthode soit transmis à l'appelé et stocké dans sa file puis que la confirmation de cette prise en compte soit renvoyée à l'appelant. Coté client, le recouvrement de l'exécution de la méthode et de la transmission du résultat est possible mais pas celui de l'envoi de la requête ni de la réception de l'accusé de réception.

L'objectif du rendez-vous est double. Premièrement, il assure à l'appelant que son appel a bien été reçu par l'appelé. Deuxièmement, il assure que l'ordre de stockage des appels dans la file d'un objet actif respecte les dépendances entre ces appels comme nous le verrons sur l'exemple de la figure 13 page suivante.

Dans le cas d'une panne de l'appelé, le rendez-vous permet de prévenir l'appelant, au moment de l'appel, que son correspondant est en panne. En fait, ce que garantit le rendez-vous, c'est le fait que l'appelé a fonctionné suffisamment longtemps pour émettre un accusé de réception de l'appel. Cela ne garantit pas que l'appelé fonctionne toujours au moment où l'accusé de réception est reçu. Si il ne fonctionne effectivement plus, l'appelant ne sera pas informé de la panne de son correspondant – survenue immédiatement après que l'accusé de réception a été envoyé – et ne pourra soupçonner cela que lorsqu'il cherchera à utiliser le résultat. Ainsi, étant donnés les aléas liés aux pannes, l'appel d'une méthode à distance n'a pas forcément pour conséquence son exécution, même lorsqu'on a la garantie de l'arrivée de cet appel. Du coup, la plate-forme E fait le choix de communications sans rendez-vous et propose un mécanisme spécifique et distinct de détection de panne. Cela implique que les appels asynchrones sur un objet *en panne* seront acceptés indéfiniment et silencieusement alors qu'ils n'auront aucune chance d'aboutir. Le mécanisme de rendez-vous de ProActive permet de prévenir l'appelant dès l'appel suivant.

Le rendez-vous garantit également que l'ordre de stockage des appels dans la file d'un objet actif respecte, non seulement l'ordre dans lequel ils ont été effectués, mais surtout les dépendances entre ces appels. Par exemple, sur la figure 13 page suivante, l'objet `a` invoque d'abord la méthode `m()` de `b`. Par la suite, il invoque une méthode sur le résultat `r` de ce premier appel qui l'oblige à attendre la fin de la méthode `b.m()` pour exécuter `r.use()` et continuer son exécution.[3] Il y a donc une dépendance entre la fin de la méthode `b.m()` et l'instruction `r.use()`. Enfin, il invoque la méthode `m2()` de l'objet

[3]Selon la plate-forme utilisée, l'attente de la fin de la méthode `b.m()` ne s'écrirait pas de la même façon. Nous avons choisi ici une syntaxe proche de ProActive. On aurait quelque chose comme `r.waitForResult()` avec Mandala ou `when(r)->done(r){c.m2()}` avec E mais le résultat serait identique. Nous ne nous étendrons pas sur les questions de syntaxe, le lecteur est renvoyé aux documentations des différentes plate-formes pour plus de détails [43, 106, 127].

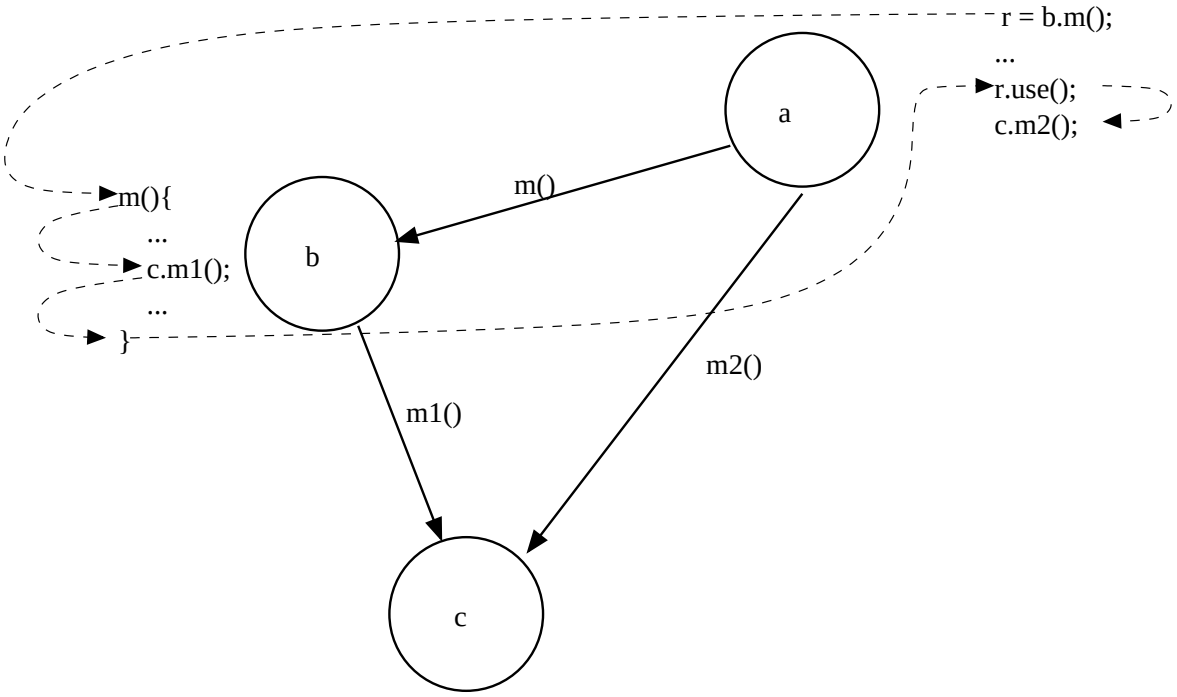


FIG. 13 – Dépendances entre plusieurs appels

`c`. De par l'attente du résultat de la méthode `b.m()`, il y a une dépendance, pour l'objet `a` entre la terminaison de l'appel `b.m()` et l'appel `c.m2()`. La méthode `m()` de l'objet `b`, quant à elle, invoque la méthode `m1()` de `c`. L'appel `c.m1()` dépend donc de l'appel `b.m()`. Ces dépendances sont représentées par les flêches en pointillé sur la figure 13. L'ordre des appels dans la file de l'objet `c` respectant ces dépendances est le suivant :

1. `c.m1()`
2. `c.m2()`

Bien entendu, cela ne correspondra pas forcément à l'ordre dans lequel ces appels seront exécutés si la file d'appel de l'objet `c` n'est pas traitée dans l'ordre *FIFO*. Cependant, le mécanisme de rendez-vous garantit, dans cet exemple, que l'ordre de stockage des appels dans la file de `c` sera celui-ci et aucun autre.

Cet ordre pourrait ne pas être respecté par une plate-forme n'utilisant pas un mécanisme de rendez-vous. C'est le cas de E ou de Mandala, selon le type des références asynchrones utilisées. En effet, sans rendez-vous la méthode `b.m()` pourrait terminer avant que son appel `c.m1()` n'ait été ajouté à la file de `c`.

Nous pensons qu'il est possible de respecter cet ordre de stockage des appels en relâchant les contraintes du rendez-vous. Nous posons comme conjecture que, dans un système d'objets actifs tel que celui proposé par ProActive, le mécanisme de communication par rendez-vous est équivalent, en ce qui concerne l'ordre des appels, à un mécanisme de communication asynchrone s'il respecte les deux conditions suivantes :

1. le lien de communication entre deux objets actifs respecte l'ordre d'émission des messages ;
2. une méthode d'un objet actif ne peut être considérée comme terminée par l'appelant que lorsque tous les appels de méthode sortants – c'est-à-dire les appels à d'autres objets actifs – déclenchés par cette méthode ont été transmis vers et reçus par les différents objets actifs cibles.

Intuitivement, tous les appels d'un objet actif vers un autre arrivent dans l'ordre dans lequel ils ont été émis. Mais les appels d'un objet actif vers plusieurs objets actifs distincts peuvent s'entremêler et les communications peuvent se dérouler de façon asynchrone par rapport à l'exécution du programme. Une méthode d'un objet actif n'est terminée que lorsque toutes les communications de cette méthode sont terminées. En fait, plutôt que de considérer l'ordre de réception des messages par un objet actif, nous considérons l'ordre de leur émission. En incluant le message permettant d'informer l'appelant que la méthode en cours est bien terminée, cette technique assure que l'ordre de stockage des appels d'un objet actif dans les files des différents appelés respecte la même relation de dépendance qu'avec un système à base de rendez-vous. Sur l'exemple de la figure 13 page ci-contre, l'attente sur l'instruction `r.use()` ne se terminera que lorsque l'appel `c.m1()` aura bien été pris en compte.

Cette approche permettrait d'augmenter le taux de recouvrement des communications entre objets actifs mais on doit noter qu'avec ProActive, les arguments entre objets actifs sont passés par copie. Dans le cas où un programme modifie une donnée après avoir demandé à la passer en argument à un objet actif, c'est la donnée avant modification qui doit être transmise. Ceci implique que même si la communication peut être faite de façon asynchrone, la copie des arguments, si on souhaite respecter ce fonctionnement, doit être faite de façon synchrone, au moment de l'appel.

4.2.2 Attente par nécessité

L'attente par nécessité permet au processus ayant appelé une méthode de façon asynchrone de se bloquer en attendant que le résultat de celle-ci soit effectivement disponible. Cette attente peut être explicite ou implicite.

Dans le cas de Mandala, l'attente par nécessité est **explicite**. Une méthode peut être appliquée sur un *futur* afin de bloquer explicitement le processus appelant jusqu'à ce que la méthode asynchrone correspondante à ce futur soit terminée et le résultat disponible. Après cette attente par nécessité, le résultat réel peut être directement utilisé.

Dans le cas de la plate-forme ProActive, l'attente par nécessité est **implicite**. Il n'est pas nécessaire de demander à attendre sur le futur. Ce dernier peut être utilisé directement comme s'il s'agissait du véritable résultat. Lors d'une telle utilisation – appel de méthode sur ce *résultat/futur* – le processus appelant est bloqué par la plate-forme le temps que la méthode asynchrone correspondante soit terminée et le résultat disponible.

Dans le cas de l'attente par nécessité explicite, le type du futur n'est pas compatible

avec celui du résultat. Avant d'utiliser ce dernier, par exemple en le transmettant à une bibliothèque, il est indispensable de déclencher une attente par nécessité. Cela introduit une sorte de barrière dans le programme que l'on ne retrouve pas dans l'attente par nécessité implicite puisque, dans ce cas, le futur peut directement être transmis à d'autres bibliothèques qui l'utiliseront de façon transparente et seront, si nécessaire, bloquées de façon transparente et au moment opportun.

Dans tous les cas, le mécanisme d'attente par nécessité bloque le processus qui effectue cette opération. Selon le modèle d'asynchronisme utilisé, cela peut être source d'étreinte fatale. Dans le cas des objets actifs de ProActive, un seul processus exécute les méthodes d'un objet actif. Supposons que ce processus, appelons-le A, se bloque en attente du résultat d'une méthode $m()$ invoquée sur un autre objet actif. Alors il ne peut plus exécuter de nouvelles méthodes tant que le résultat de `m()` n'est pas disponible. Imaginons que `m()` appelle *récursivement* une méthode sur l'objet actif de A puis attende ce résultat. Alors A n'exécutera pas cette méthode tant qu'il n'aura pas reçu le résultat de `m()` qui ne sera pas disponible tant que la méthode invoquée par `m()` n'aura pas été exécutée. Cela conduit à une étreinte fatale.

Enfin, il faut noter que, puisque le mécanisme d'attente par nécessité bloque le processus en attente d'un résultat, l'asynchronisme des appels ainsi que le recouvrement potentiel en sont réduits d'autant. Pour que cette approche soit la plus efficace possible, il est important que l'intervalle de temps entre l'appel de la méthode et l'utilisation de son résultat soit le plus grand possible. Ceci limite la durée de l'attente par nécessité et permet un degré de parallélisme plus important. Il est donc intéressant de déclencher un appel asynchrone le plus tôt possible et d'utiliser le résultat le plus tard possible. Cette démarche est contre-nature. Un développeur aura plutôt tendance à utiliser le résultat d'un appel de méthode immédiatement après cet appel et à n'effectuer un appel de méthode qu'au moment où il en a besoin. Il semble donc important d'attirer l'attention du développeur sur l'asynchronisme de ses appels afin qu'il puisse concevoir son application de façon adaptée.

4.2.3 Utilisation anticipée du résultat

Une alternative à l'attente par nécessité est l'utilisation anticipée de résultat. L'idée est d'invoquer les méthodes sur un futur de façon asynchrone, avant que le résultat correspondant ne soit disponible. Dans ce cas, plutôt que de bloquer l'appelant, ses appels sont stockés dans une file et seront exécutés lorsque le résultat sera disponible. L'asynchronisme n'est alors plus limité de la même façon qu'avec l'attente active. En outre, le risque d'étreinte fatale tel qu'il est posé avec l'attente active n'apparaît plus puisqu'aucun processus n'est bloqué.

Dans le cas de ProActive, ce type d'appel est impossible. ProActive repose sur un modèle hybride d'objets passifs et d'objets actifs. La plate-forme n'effectue un appel asynchrone que si l'objet cible est un objet actif. De plus, elle doit connaître la file associée

à cet objet pour y stocker ces appels. Savoir si le résultat futur d'un appel de méthode sera un objet actif et de quel objet actif il s'agira, est impossible dans le cas général. L'attente par nécessité est alors obligatoire.

Mandala a recours à l'attente par nécessité explicite. Il n'est donc pas possible d'invoquer une méthode du résultat, même via son futur, tant qu'une attente explicite par nécessité n'a pas été déclenchée. Mandala propose également un mécanisme événementiel permettant à l'appelant d'une méthode asynchrone de fournir un code qui sera automatiquement appelé à la fin de la méthode. Ce mécanisme événementiel pourrait permettre d'étendre Mandala pour l'enchaînement d'appels asynchrones sur un futur mais son utilisation n'est pas évidente.

E intègre le mécanisme d'appel asynchrone sur un futur. Il est possible d'invoquer directement une méthode asynchrone sur le résultat d'un autre appel asynchrone. Plutôt que de provoquer une attente par nécessité, comme dans le cas de ProActive, ces appels seront stockés dans une file jusqu'à ce que le résultat soit disponible. Cela est rendu possible dans la mesure où les méthodes de n'importe quel objet peuvent être invoquées de façon asynchrone dans E. Il n'est donc pas nécessaire de connaître le type du résultat d'un appel pour savoir que l'asynchronisme est possible.

Après une attente par nécessité, l'appelant sait que l'appel asynchrone est terminé, et s'il s'est bien déroulé ou s'il a levé une exception. L'appel asynchrone sur un futur ne fournit pas cette information. E propose une construction supplémentaire, `when`, qui permet d'effectuer un traitement lorsqu'un appel asynchrone est terminé et de traiter les exceptions éventuelles. La clause `when` fonctionne selon un modèle événementiel : le programme fournit une séquence d'instructions qui sera exécutée lorsque la méthode asynchrone sera terminée. Nous reviendrons sur cette clause section 4.4.3 page 104.

4.3 Gestion des exceptions

L'appel asynchrone de méthode à distance pose des problèmes particuliers pour la gestion des exceptions. Dans le cas synchrone, le mécanisme de gestion des exceptions est tel que, lorsqu'une méthode est appelée, si son exécution provoque la levée d'une exception, alors l'exécution de la méthode appelante est interrompue au niveau de l'appel de méthode et l'instruction suivant immédiatement cet appel de méthode n'est pas exécutée. [4] Dans le cas du langage Java, si cet appel a été effectué dans un bloc capable de gérer la survenue d'une telle exception alors l'exécution est déroutée vers le bloc de gestion de l'exception sinon la méthode appelante se termine et l'exception est propagée à la méthode ayant appelée cette dernière. Ce processus se poursuit jusqu'à ce qu'un bloc soit capable de gérer l'exception en question.

La levée d'une exception par une méthode asynchrone n'interrompt naturellement pas

[4] A notre connaissance, cette caractéristique est valable pour tous les langages proposant un mécanisme d'exceptions.

l'exécution de l'appelant qui se poursuit en parallèle de la méthode appelée. En particulier, l'instruction suivant immédiatement l'appel à la méthode est exécutée. La levée de l'exception ne se fait donc plus dans le même contexte que dans le cas synchrone et elle doit être traitée de façon spécifique.

L'approche de la plate-forme ProActive est simple : si une méthode déclare lever une exception, l'appel n'est pas asynchrone mais synchrone. De cette façon, ProActive peut se reposer sur le mécanisme classique de gestion d'exception et garantir qu'une telle exception sera bien gérée correctement. Cependant, cela limite significativement l'utilisation des exceptions dans ProActive qui font perdre l'intérêt de l'asynchronisme – qui n'existe plus dans ce cas.[5] De plus, il est possible qu'une méthode lève une exception sans que cela apparaisse dans sa signature. ProActive fera alors l'appel de façon asynchrone et l'exception ne pourra plus être gérée normalement.

Mandala clarifie les choses en spécifiant qu'une méthode asynchrone ne lève aucune exception. Seule l'attente par nécessité explicite est susceptible de lever une exception qui correspondra à l'exception levée par la méthode asynchrone correspondante. Mandala propose également des mécanismes alternatifs : transmission d'un objet *gestionnaire d'exception* lors d'un appel asynchrone (fonctionnement événementiel) ; interruption du processus léger ayant fait l'appel afin de l'avertir au plus tôt de la survenue d'une exception ; *blocage* des appels sur un objet dont une méthode a levé une exception jusqu'à confirmation de ce que l'exception a bien été gérée.

Dans le cas de E, un appel asynchrone ne lève pas non plus d'exception. C'est dans la construction *when* (Cf. section 4.2.3 page 98) que l'exception peut être gérée.

Une des particularité de la gestion des exceptions dans un système synchrone est la garantie que le flot d'exécution de l'appelant sera dérouté vers une routine de gestion d'exception dans le cas où la méthode qu'il appelle provoque la levée d'une telle exception. Ainsi, le programme ne peut ignorer la survenue d'une exception, quitte à se terminer brutalement.[6] On ne retrouve pas ce comportement avec la gestion d'exception de Mandala ou de E. Par exemple avec Mandala, si le futur d'un appel asynchrone n'est jamais utilisé, soit parce que l'état de cet appel n'est pas intéressant, soit par omission, alors la survenue d'une exception n'aura aucun effet sur l'appelant. Dans le cas de E, le problème est presque pire puisque l'appel de méthode asynchrone sur un futur n'indique rien sur l'état de la méthode correspondante et seule l'utilisation de la construction `when` permet de s'en rendre compte.

Un mécanisme de gestion des exceptions adapté aux méthodes asynchrones doit être proposé, qui propose une garantie comparable au déroutement de l'appelant du cas synchrone. Nous n'avons pas connaissance de l'existence d'un tel mécanisme.

---

[5] On peut noter la proposition d'un nouveau mécanisme de gestion des exceptions [18] permettant de repousser l'attente de l'appelant à la fin du bloc susceptible de gérer l'exception déclarée par une méthode. Mais dans ce cas, l'instruction immédiatement après celle ayant levé l'exception aura été exécutée.

[6] Dans le cas où le programmeur aura spécifié une routine de traitement d'exception *silencieuse*, celle-ci pourra être ignorée par le programme. Mais il s'agit alors d'un choix explicite du développeur, pas d'un oubli… théoriquement.

4.4 Masquage de l'asynchronisme

Chacune des trois plate-formes qui nous ont permis d'illustrer l'appel de méthode asynchrone adopte une démarche différente en ce qui concerne la visibilité de cet asynchronisme. Nous avons traité la question du masquage des communications dans les appels de méthode à distance et le problème que cela pouvait poser à la section 2.2.3 page 60. Nous n'y reviendrons pas ici mais notons tout de même que ProActive pose le même problème en passant les arguments par copie de façon transparente. Mandala également pose ce problème dans la mesure où il ne sera pas possible de distinguer entre un appel asynchrone local ou un appel asynchrone distant et où le passage des arguments sera fait par référence dans le cas local et par copie dans le cas distant. E ne pose pas ce problème puisque les arguments sont toujours passés par référence sauf pour les constantes.

4.4.1 Asynchronisme transparent

L'un des objectifs de la plate-forme ProActive est la réutilisation de code existant. Pour cela, ProActive propose un mécanisme transparent d'appel asynchrone. Syntaxiquement, un appel asynchrone a la forme d'un appel synchrone. En pratique, l'appelant n'est pas bloqué pendant l'exécution. Cette transparence est assurée grâce à un certain nombre de principes. Tout d'abord, un objet actif, c'est-à-dire un objet dont les méthodes sont appelées de façon asynchrone, est compatible[7] avec sa version non active. Ainsi, un code existant utilisant un objet d'un type donné sera *syntaxiquement* compatible avec la version active de ce même objet. Ensuite, un appel de méthode ne renvoie pas le résultat attendu mais un *futur* sur ce résultat. Ce futur est compatible avec le véritable résultat attendu. C'est-à-dire que *syntaxiquement*, il peut être utilisé en lieu et place du véritable résultat dans le programme.

Ce masquage syntaxique de l'aspect asynchrone d'un appel oblige à mettre en œuvre un mécanisme transparent de gestion des exceptions en asynchrone. Pour cela, nous avons vu que le choix de ProActive est de rendre synchrone un appel pouvant lever une exception. Cependant, une méthode peut lever une exception sans que cela apparaisse au niveau de sa signature. Dans ce cas, l'appel est fait de façon asynchrone et la gestion de l'exception n'est plus transparente puisqu'elle se fera au moment de l'attente par nécessité.

La sémantique synchrone de gestion des exceptions veut que l'instruction qui suit immédiatement l'appel d'une méthode ayant levé une exception ne soit pas exécutée. Si on souhaite gérer de façon totalement transparente la survenue d'une exception lors d'un appel asynchrone, il faut respecter cette sémantique et cela implique d'être capable d'annuler l'effet de l'exécution d'une partie du programme. En particulier, toutes les instructions exécutées par l'appelant entre un appel asynchrone de méthode et la survenue d'une exception au cours de cet appel doivent être annulées afin de remettre le programme dans l'état dans lequel il se trouvait au moment de cet appel. On peut envisager différentes

[7] Cette compatibilité est assurée au niveau du langage par l'exploitation du polymorphisme.

méthodes pour permettre l'annulation d'instructions, citons-en deux :

1. les points de reprise, que nous avons vus en 1.1.2 page 11. L'état du programme est sauvegardé à intervalles réguliers. Lorsqu'on souhaite annuler un certain nombre d'instructions, on remet le programme dans l'état dans lequel il était au premier point de reprise avant la dernière instruction à annuler et on reprend l'exécution jusqu'à atteindre l'instruction souhaitée ;

2. l'exécution inverse. Chaque instruction du langage possède son instruction inverse. Des données peuvent être enregistrées dans un historique pour pouvoir exécuter certaines instructions inverses. L'opération +, par exemple, effectue l'addition de deux entiers et retourne un entier résultat. L'opération inverse est alors la soustraction. Cependant, dans le cas de Java, par exemple, l'opération + *détruit* ses deux opérandes. Il est alors nécessaire de stocker l'un de ces deux opérandes dans un historique. Lorsqu'on souhaite annuler une addition, il suffit de soustraire au résultat de l'addition l'opérande stocké dans l'historique pour retrouver le deuxième opérande. De même, il est nécessaire de pouvoir retrouver, lors d'une exécution inverse, l'instruction précédent l'instruction courante dans le cas où il se serait produit un saut lors de l'exécution normale. Là encore, ceci implique l'enregistrement d'informations dans un historique. Dans le cas des boucles, la quantité d'informations à enregistrer risque d'être relativement importante.

Le caractère destructif de la plupart des instructions usuelles d'un langage nécessite le stockage d'une quantité importante de données pour leur annulation, sans parler de la gestion de la mémoire, puisque des opérations de ramasse-miettes pourraient interdire l'annulation de certaines instructions. Les points de reprise, dans le cas de la plate-forme ProActive sont déjà utilisés pour la gestion transparente des pannes partielles, comme nous l'avons vu en 1.1.2 page 11. Cependant, leur utilisation pour la gestion des exceptions en feraient un élément constitutif du paradigme de programmation ProActive alors que ce n'est aujourd'hui qu'un outil optionnel. De façon générale, ces mécanismes se heurtent à l'interfaçage avec l'extérieur : comment annuler l'accès à un fichier, l'écriture d'une donnée sur une `socket`, etc ? Pour toutes ces raisons, les mécanismes d'annulation d'instructions sont difficiles à mettre en œuvre et souvent limités au cadre du débogage. On pourra trouver un tel exemple pour le langage Java dans [32]. Il semble plus raisonnable de proposer une alternative pour la gestion des exceptions qui soit spécifique aux appels asynchrones.[8]

Enfin, la plate-forme ProActive repose sur l'attente par nécessité qui, comme nous l'avons vu en 4.2.2 page 97 nécessite une anticipation maximale des appels asynchrones et une utilisation le plus tard possible de leur résultat. C'est une démarche plutôt contre-nature lors de l'écriture d'un programme séquentiel et qui n'est donc pas facilitée par le masquage de l'asynchronisme.

[8] C'est d'ailleurs l'approche suivie par le nouveau mécanisme de gestion des exceptions de ProActive [18] que nous avons déjà évoqué dans la note de bas de page numéro 5 page 100.

4.4.2 Asynchronisme semi-transparent

L'objectif de Mandala est de proposer une syntaxe pour les appels de méthode asynchrones qui soit différente mais proche de celle des appels synchrones classiques. Une méthode asynchrone aura une signature proche de celle de sa version synchrone : mêmes arguments et même nom, préfixé par `rami_`. Le type du retour, par contre, sera différent ; il s'agira explicitement d'un futur, incompatible avec le type de retour de la méthode synchrone. Enfin, la méthode asynchrone ne lèvera aucune exception. Un compilateur nommé *jayac* permet de générer la version asynchrone d'une classe. A titre d'exemple, on considère ici un extrait de la classe `java.io.BufferedReader` :

```
1  package java.io;
2  public class BufferedReader extends Reader{
3      ...
4      String readLine() throws IOException{
5          ...
6      }
7      ...
8  }
```

Le programme suivant présente un extrait de la classe générée par jayac à partir de `java.io.BufferedReader`[9] :

```
1  package jaya.java.io;
2  public class BufferedReader extends Reader{
3      ...
4      FutureClient rami_readLine(){
5          ...
6      }
7  }
```

Le développeur qui souhaite invoquer une méthode de façon asynchrone avec Mandala ajoute simplement le préfixe `rami_` à son appel de méthode. Grâce à cette syntaxe, ce dernier est conscient de l'aspect asynchrone de son appel – préfixe `rami_` et renvoi d'un futur. Il peut donc, en toute connaissance de cause, effectuer ses appels asynchrones le plus tôt possible et n'utiliser les résultats que le plus tard possible. L'aspect asynchrone est également clairement apparent en ce qui concerne les exceptions, comme nous avons pu le voir en 4.3 page 99. L'objectif de la semi-transparence est de limiter le masquage, en particulier en faisant apparaître clairement la notion de futur et en évitant la compatibilité entre futur et véritable résultat. Ce dernier point oblige à une attente par nécessité explicite et augmente la conscience du développeur. La contrepartie, comme nous l'avons vu en 4.2.2 page 97, est que l'attente par nécessité explicite peut constituer une barrière gênante pour le programme.

On remarquera pour finir que le mécanisme de semi-transparence de Mandala fait apparaître clairement l'aspect asynchrone des appels. Cependant, le type d'asynchronisme

[9] Il faut noter que, par soucis de clarté, nous ne présentons pas toutes les méthodes générées par *jayac* à partir de la méthode `readLine` originale.

utilisé est masqué. Syntaxiquement, il n'est pas possible pour le développeur de savoir quelle politique asynchrone va être utilisée par son appel de méthode. En particulier dans le cas d'appels asynchrones à distance, rien ne différencie syntaxiquement un appel de méthode asynchrone par rendez-vous d'un appel totalement asynchrone. En pratique, si le bon fonctionnement d'un programme dépend directement de la politique asynchrone de telle ou telle référence, c'est au développeur de s'assurer que cette politique est bien celle attendue.

4.4.3 Asynchronisme explicite

E propose un asynchronisme explicite. La syntaxe fait apparaître clairement l'aspect asynchrone des appels et le type de cet asynchronisme est unique et clairement défini. Sur le plan syntaxique, un appel asynchrone se distingue d'un appel synchrone par l'utilisation de `<-` en lieu et place du point pour l'appel de méthode. Dans l'exemple suivant, l'appel à la méthode `m()` de l'objet `o` est synchrone, l'appel à la méthode `n()` est asynchrone :

```
1 r1 := o.m()
2 r2 := o<-n()
```

L'objet `r2` sera donc un futur – *promise*, dans le vocabulaire E – et, à la différence de `r1`, les appels synchrones seront interdits sur cet objet. Dans la suite de l'exemple, l'appel asynchrone sur *r2* est possible et non bloquant même si la méthode `n()` n'est pas encore terminée et que le résultat associé n'est pas disponible. C'est l'utilisation anticipée du résultat que nous avons vu en 4.2.3 page 98 :

```
1 r1.f()
2 r2<-p()
```

Enfin, l'utilisation synchrone de `r2` ne sera possible que lorsqu'il sera certain que celui-ci aura bien été calculé. Cela n'est possible qu'au sein d'une construction `when` :

```
1 when(r2) -> done(r2){
2     r2.p()
3 }catch e{
4     println(``problème !'')
5 }
```

E propose une vision totalement explicite de l'asynchronisme. Pourtant, au niveau syntaxique, un futur est compatible avec le résultat attendu. Cette compatibilité est en fait la conséquence de l'absence de typage dans le langage E. Cela permet d'invoquer de façon transparente des méthodes asynchrones, que ce soit sur un objet ou sur un futur.

4.5 Transparence sémantique de l'asynchronisme

Finalement, dans chacune de ces plate-formes, la sémantique des appels asynchrones doit être prise en charge par le développeur. Même dans le cas où l'asynchronisme est syntaxiquement masqué, comme avec ProActive, le développeur doit s'assurer de la cohérence de son application et du déterminisme dans les interactions entre objets actifs. On pense notamment à l'exemple de la figure 13 page 96, non déterministe avec E, déterministe avec ProActive. Il est donc nécessaire d'assister le programmeur dans la prise en charge de cet aspect sémantique. Nous présentons au chapitre suivant la propriété d'activabilité définie dans le cadre de la plate-forme ProActive [9] qui permet de masquer les aspects sémantiques de l'asynchronisme. Nous verrons l'extension que nous avons proposé de cette propriété aux programmes à processus légers et sa validation. Enfin, nous décrirons les bouquets d'activations qui constituent une perspective privilégiée pour nos travaux.

Chapitre 5

Contribution à la prise en charge de la latence : activabilité étendue et bouquets d'activations

Nous avons présenté au chapitre précédent différentes plate-formes pour l'appel asynchrone de méthode à distance. Nous avons vu les différents niveaux de transparence syntaxique qu'on pouvait rencontrer. Aucune des ces plate-formes n'offre de transparence sémantique de l'asynchronisme. Autrement dit, les conséquences de l'introduction d'asynchronisme dans une application doivent être prises en charge par le développeur.

L'objectif de la propriété d'activabilité que nous présentons dans ce chapitre est justement de proposer une transparence sémantique de l'asynchronisme. Un objet qui respecte la propriété d'activabilité dans un programme peut être rendu actif, au sens ProActive du terme, sans modifier la sémantique du programme.[1] Cela offre une transparence sémantique de l'asynchronisme.

Nous avons vu précédemment les objets actifs ainsi que leur intérêt. Pour rappel, un objet actif possède son propre environnement, c'est-à-dire qu'il est le seul a accéder à l'ensemble des objets passifs qu'il utilise. De plus, les appels de méthode effectués vers un objet actif sont asynchrones ce qui permet de diminuer l'impact de la latence lorsque ces appels sont distants. L'introduction d'objets actifs dans un programme présente donc certains intérêts en ce qui concerne la distribution de ce programme et le parallélisme induit.

Le but de la propriété d'activabilité est de décrire les objets d'un programme qui peuvent être rendus actifs sans que la sémantique du programme ne soit modifiée – on les appellera objets activables. Une propriété d'activabilité est proposée dans la thèse de Romain Guider [65]. Dans ce cadre, l'ordre de traitement des appels de méthode est l'ordre

[1] Dans la suite de ce chapitre, lorsque nous disons d'une transformation qu'elle ne modifie pas la sémantique du programme, nous insinuons que le programme transformé est en bisimulation faible avec le programme d'origine (cf. section 5.3 page 118).

FIFO. Nous allons maintenant présenter cette propriété[2] ainsi que ses limites. Nous présenterons ensuite notre extension de la propriété d'activabilité qui tente de lever certaines de ces limites avant de prouver sa validité. Par la suite nous nous intéresserons à la mise en pratique de cette propriété et aux difficultés que cela peut poser. Enfin, nous présenterons les bouquets d'activations qui constituent la suite logique de nos travaux sur l'activabilité.

5.1 La *TB*-activabilité

Nous présentons la propriété d'activabilité décrite dans la thèse de Romain Guider [65, p. 127]. Dans le cadre de cette propriété, les appels de méthode sont traités dans l'ordre de leur réception. Nous utiliserons par la suite la notion informelle d'*instant d'exécution du programme*. Cela correspond à la notion formelle d'*état* utilisée dans [65] et que nous ne décrirons pas ici. De façon générale, les définitions présentées dans cette section sont reprises de [65] ; on pourra s'y référer pour plus de précisions.

Cette propriété d'activabilité s'appuie sur la notion d'accessibilité, elle même définie grâce au graphe d'accessibilité. Un objet spécial – r_s – est introduit pour représenter la méthode `main()` de l'application, c'est-à-dire la première méthode de l'application. Cet objet référence l'ensemble des objets accessibles uniquement depuis les variables locales de la méthode `main()`.

Définition 5.1. *Graphe d'accessibilité*
Le graphe d'accessibilité d'un programme à un instant de son exécution est tel que :

- *l'ensemble des nœuds du graphe est l'ensemble des objets existant à cet instant dans le programme ainsi que le nœud spécial r_s.*
- *il existe une arête du nœud o au nœud o' dans le graphe d'accessibilité si et seulement si :*
  - *o référence o' via une variable d'instance ou*
  - *un appel de méthode dans la pile référence o via sa variable $this$ et o' est référencé via une variable locale quelconque de cet appel ou*
  - *$o = r_s$ et une variable locale de la méthode `main()` référence o'.*

La figure 14(a) page suivante représente l'état d'un programme à un instant donné de son exécution. La pile est représentée à gauche de la figure. La méthode `main()` a naturellement été la première appelée puis elle a appelé la méthode `M1()`, elle même a invoqué `M2()` qui a invoqué `M3()`. Les flèches représentent des références. Les flèches sortant de la pile représentent des références via des variables locales (celles en gras étiquetées `this` référencent l'objet sur lequel la méthode a été invoquée) et les flèches entre objets représentent des références via des champs. Dans le graphe d'accessibilité correspondant – figure 14(b) page ci-contre – les sommets du graphe sont les objets du

[2] Afin de la distinguer de l'extension que nous en proposons, nous l'appellerons parfois par la suite *TB*-activabilité, en référence au terme anglais de *tree-based activability* proposé par Denis Caromel.

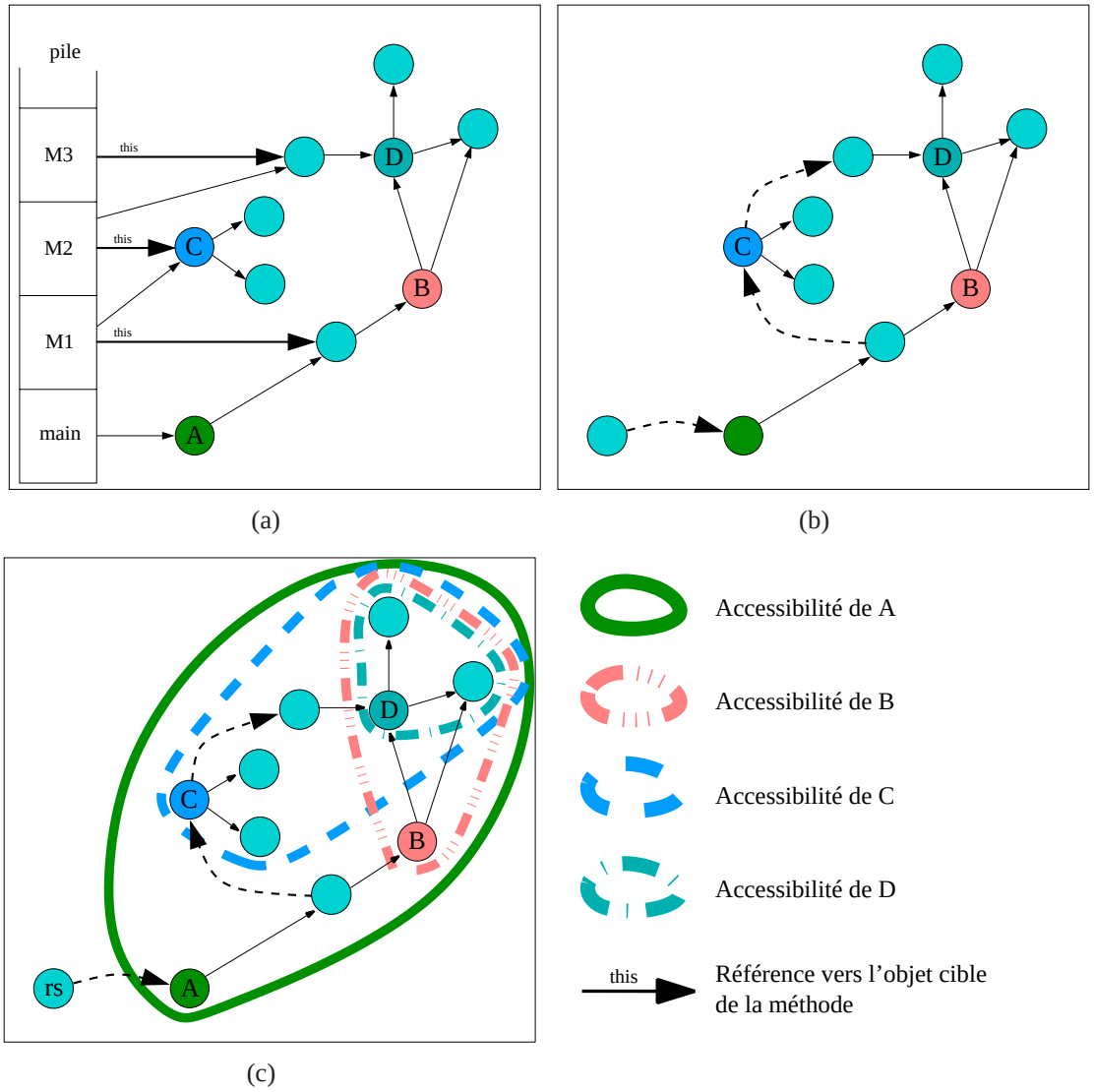


FIG. 14 – Graphe d'accessibilité.

programme. On constate que les références entre objets sont conservées sous la forme d'arêtes entre ces objets. Des arêtes supplémentaires sont ajoutées entre deux objets si une méthode invoquée sur le premier référence le second via une variable locale. Ces nouvelles arêtes sont représentées en pointillés sur la figure. Enfin, le nœud spécial r_s a été ajouté.

Définition 5.2. *Accessibilité*
L'accessibilité d'un objet o à un instant de l'exécution d'un programme est constituée de l'ensemble des objets accessibles transitivement depuis o dans le graphe d'accessibilité correspondant à cet instant de l'exécution du programme.

La notion d'accessibilité est transitive, si un objet est dans l'accessibilité d'un second alors tous les objets dans l'accessibilité du premier sont également dans l'accessibilité du second. La figure 14(c) page précédente met en évidence l'accessibilité de certains objets du graphe d'accessibilité, en l'occurrence les objets nommés A, B, C et D. On peut d'ailleurs constater que les accessibilités de B et C ne sont pas disjointes mais qu'aucune des deux n'est inclue dans l'autre. Nous verrons que c'est une condition suffisante pour que B et C ne puissent être activables.

Définition 5.3. *Activabilité*
Un objet o est activable si et seulement si, à chaque instant de l'exécution du programme, l'accessibilité de o est une partie disjointe du graphe d'accessibilité : pour tout objet o' dans l'accessibilité de o, pour tout objet o'',

- *soit o'' appartient à l'accessibilité de o,*
- *soit o appartient à tous les chemins depuis o'' jusqu'à o'.*

On peut déduire de cette définition qu'un objet activable est une articulation[3] ou une feuille[4] du graphe d'accessibilité.

Ainsi, selon cette définition, les objets B et C de la figure 14 page précédente ne sont pas activables : ce ne sont pas des articulations du graphe d'accessibilité. Enfin, l'objet D n'est pas activable non plus puisqu'il existe un chemin de B vers un objet dans l'accessibilité de D ne passant pas par D. L'objet A, par contre, est activable : tous les chemins de l'extérieur vers l'intérieur de son accessibilité passent par lui.

5.1.1 Validité de la *TB*-activabilité

La validité de l'activabilité présentée dans la thèse de Romain Guider [65] repose sur la structure arborescente du graphe des sous-systèmes obtenus par l'activation d'objets activables et sur le respect des conditions de Bernstein que nous décrirons plus loin.

[3] La suppression de ce sommet dans le graphe d'accessibilité découpe le graphe en deux composantes connexes distinctes.

[4] Un sommet sans arête sortante.

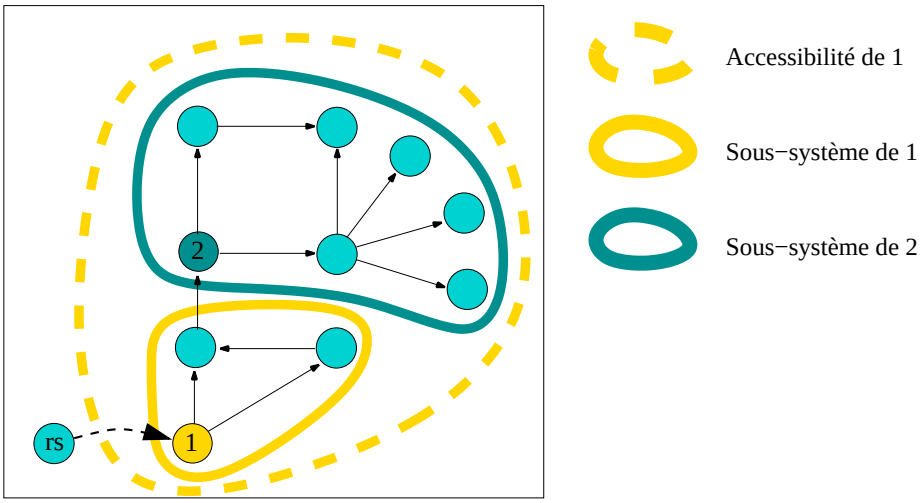


FIG. 15 – Exemple de sous-systèmes

Arborescence des sous-systèmes

Définition 5.4. *Sous-système*
Le sous-système d'un objet actif est constitué de l'ensemble des objets appartenant à son accessibilité, privé de l'ensemble des objets actifs de cette accessibilité ainsi que de leurs accessibilités respectives.

Autrement dit, le sous-système d'un objet actif est constitué de l'ensemble des objets passifs qu'il utilise. S'il utilise un autre objet actif alors cet autre objet actif ainsi que le sous-système de ce dernier ne font pas partie de son sous-système. Sur la figure 15, les objets 1 et 2 sont actifs. Leurs sous-systèmes respectifs sont représentés. On constate que, bien que 2 soit dans l'accessibilité de 1, il ne fait pas partie de son sous-système car il est lui-même actif.

Définition 5.5. *Graphe des sous-systèmes*
Pour chaque objet activable dans le graphe d'accessibilité, il existe un sommet dans le graphe des sous-systèmes représentant cet objet et son sous-système. Il existe une arête du sommet a vers le sommet b du graphe des sous-systèmes si il existe une arête d'un objet du sous-système a vers l'objet activable du sous-système b dans le graphe d'accessibilité.

La figure 16 page suivante présente un exemple de graphe d'accessibilité et le graphe des sous-systèmes correspondant. Si on s'intéresse au graphe des sous-systèmes, on constate [65] que ce graphe est en fait un arbre. On peut d'ailleurs noter que la décomposition d'un graphe en composantes biconnexes[5] est unique et arborescente. Dans la mesure où les objets activables sont des articulations ou des feuilles du graphe d'accessibilité, la structure arborescente du graphe des sous-systèmes correspondant semble naturelle.

[5] Pour tout sommet d'une composante biconnexe, la composante obtenue en supprimant ce sommet reste connexe. Les sommets communs à au moins deux composantes biconnexes sont les points d'articulation du graphe.

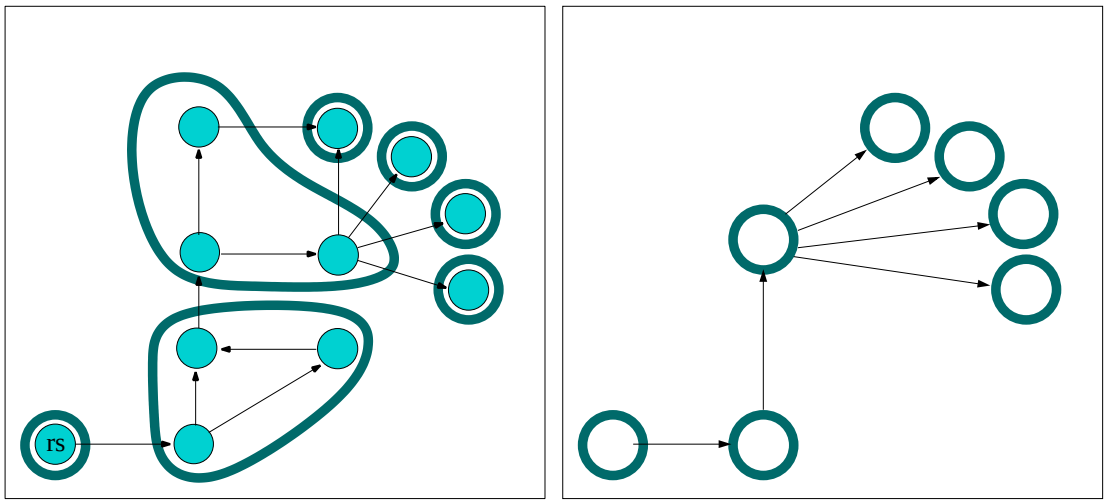


FIG. 16 – Graphe des sous-systèmes

Respect des conditions de Bernstein

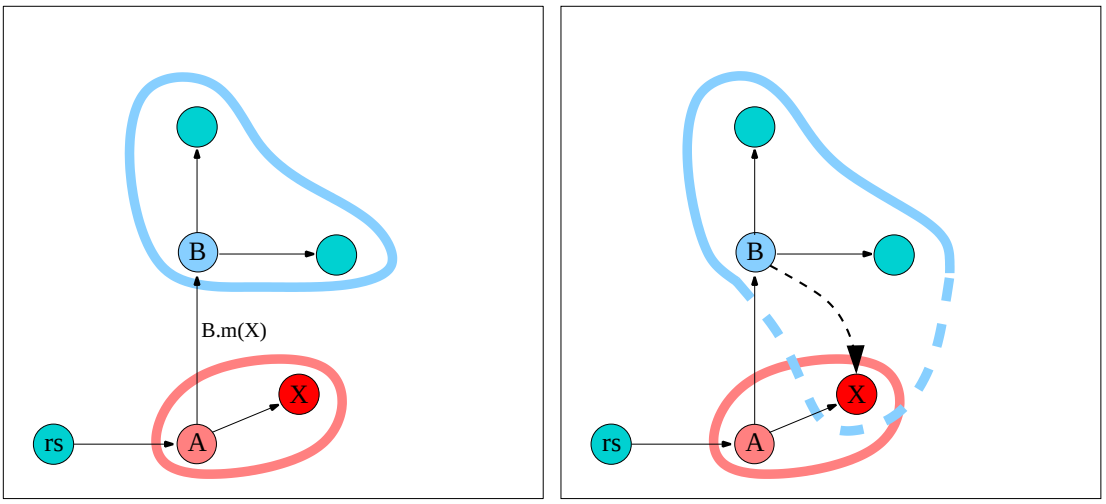
Les conditions de Bernstein spécifient que plusieurs processus sont indépendants – l'effet de leur exécution est le même quelque soit l'ordre dans lequel ils sont exécutés ou s'ils sont exécutés en parallèles – si, *pour chaque processus*, l'ensemble des données lues ou modifiées par ce processus est disjoint de l'ensemble des données modifiées par les autres processus.

La structure arborescente garantit que les appels aux méthodes de sous-systèmes *frères* sont indépendants. C'est le cas par exemple des sous-systèmes feuilles de la figure 16. Entre un *père* et un *fils*, par contre, l'indépendance est moins évidente. Nous reviendrons sur ce point dans la section suivante, page 115.

5.1.2 Limites de la *TB*-activabilité

Processus légers La *TB*-activabilité ne prend pas en compte la gestion de programmes Java contenant des processus légers. Cette démarche est raisonnable dans la mesure où l'objectif est d'introduire de la concurrence dans une application séquentielle et que le modèle des objets actifs à la ProActive sur lequel cette propriété s'appuie est une alternative aux processus légers. Cependant, cela a également pour effet de diminuer le nombre d'applications cibles, de nombreuses bibliothèques Java de base utilisant les processus légers.

Mobilité Ce critère d'activabilité interdit la mobilité des objets passifs entre sous-systèmes. En effet, selon le critère de *TB*-activabilité, un objet passif doit, à tout instant, être dans un et un seul sous-système or un sous-système est défini comme l'ensemble



(a) (b)

FIG. 17 – La mobilité entre sous-systèmes est incompatible avec la *TB*-activabilité.

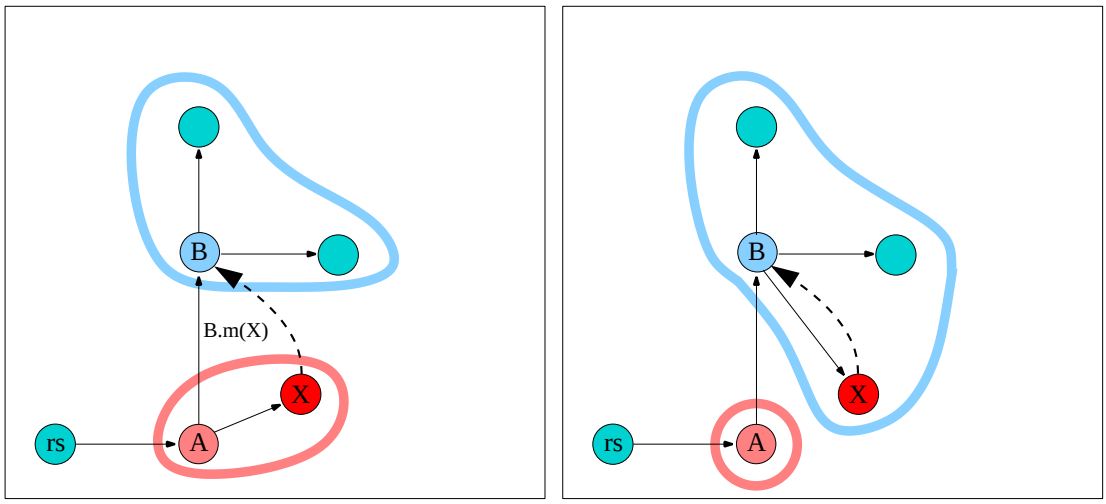
des objets passifs accessibles par un objet actif. Le transfert d'un objet passif d'un sous-système à un autre nécessite qu'à un instant du programme il soit référencé par ces deux sous-systèmes, ce qui est en contradiction avec la *TB*-activabilité. Par exemple sur la figure 17(a), les objets A et B semblent activables. Pourtant le passage d'un objet – ici X – en argument à une méthode de B par A empêche B d'être *TB*-activable : à un instant du programme – représenté figure 17(b) – B référence X, lui-même référencé par A. Pourtant, dans le cas où X n'est plus jamais utilisé par A après cet appel, l'activabilité de B serait logiquement possible. C'est le cas, par exemple, dans le pseudo-code suivant pour l'objet A :

```
1    ...
2    X = new XClass();
3    B.m(X);
4    X = null;
5    ...
```

Cette limitation implique que tout objet dont certaines méthodes reçoivent d'autres objets en argument ou retournent des objets en résultat n'est pas *TB*-activable dès qu'une de ces méthodes peut être appelée depuis un autre sous-système[6] : seuls des types primitifs peuvent être échangés entre sous-systèmes. On peut noter une possibilité d'extension simple qui permettrait le partage d'objets constants entre sous-systèmes [65].

Conditions de Bernstein Les conditions de Bernstein imposent que les ensembles des données lues et écrites par différents processus soient disjoints pour que ces processus

[6]On emploi ici un raccourci en parlant d'un *autre* sous-système puisque, notre objet n'étant pas *TB*-activable, il ne possède pas de sous-système propre à ce moment du discours.



(a) (b)

FIG. 18 – Mobilité de X entre deux sous-systèmes.

puissent être considérés comme indépendants. Ainsi les conditions de Bernstein n'interdisent pas le partage de données entre processus. Le fait qu'il existe, pour un processus, un moyen d'accès potentiel à une donnée modifiée par un autre processus n'entre pas en contradiction avec les conditions de Bernstein mais c'est l'accès, en lui-même, qui est contradictoire. Le fait, qu'à un instant donné, un objet passif soit dans deux sous-systèmes distincts n'est pas, en soit, en contradiction avec les conditions de Bernstein pourvu qu'un seul de ces deux sous-systèmes utilise réellement l'objet en question. Autrement dit, la mobilité entre sous-systèmes n'est pas en contradiction avec les conditions de Bernstein et pourrait être ajoutée à la *TB*-activabilité sans modifier sa compatibilité avec les conditions de Bernstein.

Pourtant l'ajout de la mobilité dans la *TB*-activabilité peut être source d'étreintes fatales. Considérons par exemple le pseudo-code suivant :

```
1  ...
2  X = new XClass(B);
3  B.m(X);
4  X = null;
5  ...
```

Le passage de l'objet X d'un sous-système à un autre est représenté figure 18. Conceptuellement, l'objet B est bien *TB*-activable sur les figures 18(a) et 18(b).[7] Supposons maintenant que B soit rendu actif. Sur la figure 18(a), l'objet X référence B de l'extérieur de son sous-système. Ainsi, tout appel effectué par X sur B est asynchrone. Cet appel est ajouté à la file des appels sur B puis exécuté lorsque le processus associé à B est disponible pour l'exécuter. Lorsque cet objet X passe dans le sous-système de B –

[7] Il existe un état entre les deux dans lequel B n'est pas *TB*-activable mais nous l'ignorons pour étudier l'impact de la mobilité entre sous-systèmes sur la *TB*-activabilité.

figure 18(b) page ci-contre – il possède toujours une référence sur B. On voit apparaître un cycle : B référence X qui référence B. Les appels effectués de X à B doivent maintenant, et impérativement, être synchrones. En effet, les méthodes de X sont maintenant exécutées par le processus associé à B. Si X invoque une méthode sur B de façon asynchrone puis attend le résultat de cet appel alors une étreinte fatale apparaît, qui n'existait pas dans le programme séquentiel original : le processus de B ajoute un appel de méthode à la liste de ses appels à traiter. Cet appel ne sera pas traité tant que l'appel en cours ne sera pas terminé or l'appel en cours ne sera pas terminé tant que l'appel en attente ne sera pas lui-même terminé. Cette étreinte fatale peut être évitée à la condition que le mode de communication entre X et B devienne synchrone lorsque X entre dans le sous-système de B.

Bien que la mobilité d'objets entre sous-systèmes n'entre pas en contradiction avec les conditions de Bernstein, elle entre en contradiction avec la *TB*-activabilité et doit être traitée spécifiquement. D'autre part, il parait clair que deux objets activables *frères* dans l'arbre des sous-systèmes ne partagent aucune donnée et les appels de méthode sur ceux-ci peuvent donc s'effectuer de façon concurrente conformément aux conditions de Bernstein. Cependant, dans le cas d'objets activables *père* et *fils*, un partage de données existe. En effet, le *père* modifie la file des appels du *fils* (lorsqu'il invoque une méthode du *fils*) que le *fils* lit (avant d'exécuter la méthode) et réciproquement le *fils* met à jour les futurs que le *père* utilise. Pour ces deux raisons, le respect des conditions de Bernstein par le critère de *TB*-activabilité ne semble pas évident. Sans remettre en cause la validité du critère lui-même, cela semble indiquer qu'une preuve plus complexe est nécessaire. On pourra d'ailleurs consulter la preuve par induction proposée par Isabelle Attali, Denis Caromel et Romain Guider dans [8].

5.2 Extension de l'activabilité

La *TB*-activabilité n'autorise pas la mobilité des objets entre sous-systèmes ni l'utilisation des processus légers dans une application. Cela signifie qu'elle ne s'applique à aucune application existante qui ferait usage des processus légers. Dans un langage comme Java, cela peut représenter une restriction importante dans la mesure où un certain nombre de bibliothèques de base utilisent les processus légers, en particulier dans le domaine des entrées-sorties ou des interfaces graphiques. L'interdiction de la mobilité des objets entre sous-systèmes est également une limitation importante puisqu'elle empêche l'activation de pratiquement tout objet dont les méthodes reçoivent en argument ou retournent comme résultat un objet. Nous proposons donc l'extension de la *TB*-activabilité [25].

Conceptuellement cette extension respecte le principe de base de la *TB*-activabilité : pour qu'un objet soit activable, à tout instant du programme, tous les chemins de l'extérieur vers l'intérieur de son accessibilité doivent passer par lui. De plus :

- il ne doit être utilisé que par un seul processus léger à la fois ;
- tout objet retourné comme résultat d'un appel d'une de ses méthodes quitte immédiatement son accessibilité ;
- enfin, tout objet passé en argument à l'une de ses méthodes quitte immédiatement le sous-système de l'appelant.

De plus, tout comme pour la *TB*-activabilité, cette extension suppose que les appels de méthode sur un objet actif sont traités dans l'ordre FIFO.

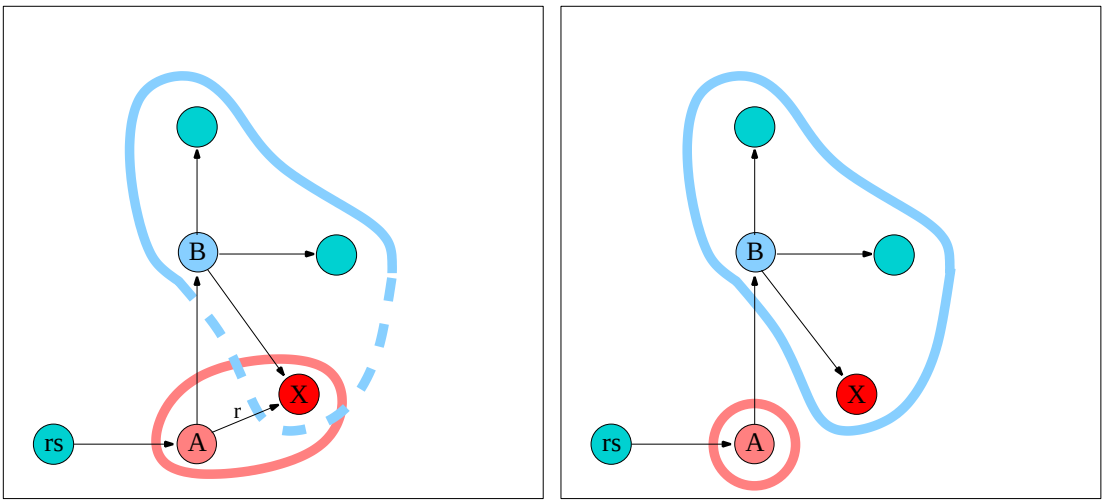
Un appel de méthode sur un objet actif est asynchrone et est exécuté par le processus associé à cet objet. Ainsi, l'accès simultané à un objet actif par plusieurs processus légers peut provoquer des étreintes fatales et des incohérences dans le comportement du programme en raison des mécanismes de synchronisation qu'ils pourraient mettre en œuvre. On peut considérer, par exemple, les méthodes `wait()` et `notify()` d'un objet, permettant respectivement de bloquer un processus léger et débloquer un processus léger précédemment bloqué. Dans le cas où plusieurs processus légers utiliseraient simultanément un objet actif en ayant recours à ces méthodes alors, d'une part, on verrait une incohérence apparaître puisque le processus léger appelant la méthode `wait()` ne serait pas bloqué (appel asynchrone) et d'autre part c'est le processus de l'objet actif qui devrait à la fois se bloquer (en exécutant effectivement la méthode `wait()`) et à la fois se débloquer, on aboutirait alors à une étreinte fatale. En imposant qu'un objet activable ne soit utilisé que par un seul processus léger *à la fois* on évite ce type de situation. Ceci dit, cela n'interdit en rien le sous-système d'un tel objet actif d'être, lui-même, composé de différents processus légers.

En ce qui concerne la mobilité des objets passifs entre sous-systèmes, nous modifions la définition du graphe d'accessibilité pour qu'elle intègre la notion d'utilisation potentielle.

Définition 5.6. *Potentiel d'utilisation*
A un instant de l'exécution d'un programme, il existe un potentiel d'utilisation d'un objet o via une référence r si :

- *une méthode de o invoquée via la référence r est en cours d'exécution ou*
- *dans la suite du programme, une méthode est susceptible d'être invoquée sur o via la référence r ou*
- *dans la suite du programme, une référence r' sur l'objet o peut être construite par copie de la référence r.*

Le potentiel d'utilisation s'applique aux références et nous permet d'éliminer celles qui ne seront plus utilisées dans la suite du programme. A partir d'un instant de l'exécution, nous ne nous intéresserons plus qu'aux références utilisées par la suite pour invoquer une méthode sur l'objet référencé à cet instant ou pour transmettre une référence sur cet objet à une autre partie du programme. On peut noter que si le seul potentiel d'utilisation d'une référence r est sa copie en une référence r' pour laquelle il n'existe pas de potentiel d'utilisation, on considérera malgré tout que r est potentiellement utilisée : la définition



(a) `B.m(X) ; X.f() ;` (b) `B.m(X) ; X = null ;`

FIG. 19 – Mobilité ou partage de X entre deux sous-systèmes.

que nous donnons nous permet déjà d'accepter la mobilité d'objets entre sous-systèmes et sa vérification en pratique n'est pas triviale ; nous évitons donc de complexifier cette définition. On considérera maintenant qu'un objet *quitte* un sous-système lorsque cet objet n'est potentiellement plus utilisable par aucune référence de ce sous-système. Nous redéfinissons le graphe d'accessibilité pour prendre en compte cette notion de potentiel d'utilisation :

Définition 5.7. *Graphe d'accessibilité*
Le graphe d'accessibilité d'un programme à un instant de son exécution est tel que :

- *l'ensemble des nœuds du graphe est l'ensemble des objets existant à cet instant du programme ainsi que le nœud spécial r_s.*
- *il existe une arête du nœud o au nœud o' dans le graphe d'accessibilité si et seulement si :*
  - *o référence o' via une variable d'instance et il existe un potentiel d'utilisation de o' via cette variable d'instance ou*
  - *un appel de méthode dans la pile référence o via sa variable $this$ et o' est référencé via une variable locale quelconque de cet appel et il existe un potentiel d'utilisation de o' via cette variable locale ou*
  - *$o = r_s$ et une variable locale de la méthode `main()` référence o' et il existe un potentiel d'utilisation de o' via cette variable locale.*

La figure 19 présente les configurations après l'appel à la méthode `m()` de l'objet B par l'objet A avec X en argument. Sur la figure 19(a), il existe un potentiel d'utilisation de X via la référence r. Ainsi X est accessible par B mais n'a pas quitté le sous-système de A. Il existe donc un chemin de l'extérieur (A) vers l'intérieur (X) de l'accessibilité de B ne passant pas par B. B n'est donc pas activable. Sur la figure 19(b), au contraire, X quitte

le sous-système de A : il n'existe pas de potentiel d'utilisation de X via r qui n'apparaît donc pas dans le graphe d'accessibilité. X est donc uniquement dans l'accessibilité de B ce qui permet à B, dans le cas où toutes les autres conditions requises seraient respectées, d'être activable.

Définition 5.8. *Activabilité*
Un objet o est activable si et seulement si :

1. *à chaque instant de l'exécution du programme, pour tout objet o' dans l'accessibilité de o, pour tout objet o'',*
   - *soit o'' appartient à l'accessibilité de o,*
   - *soit o appartient à tous les chemins depuis o'' jusqu'à o'*
2. *les méthodes de l'objet o ne sont invoquées, depuis l'extérieur de son sous-système, que par un seul processus léger.*

Ainsi, les méthodes d'un objet activable ne peuvent être invoquées que par un seul processus léger mais le sous-système correspondant à cet objet peut, lui-même, être composé de plusieurs processus légers. D'autre part, les objets peuvent se déplacer de sous-systèmes en sous-systèmes grâce à la nouvelle définition du graphe d'accessibilité.

5.3 Validité de l'activabilité étendue

L'activabilité étendue offre de nouvelles possibilités en terme de réutilisation de code – les applications à plusieurs processus légers peuvent être étudiées – ainsi qu'en terme de souplesse puisqu'il est maintenant possible, dans les conditions définies précédemment, d'activer un objet dont les méthodes reçoivent en argument ou retournent en résultat des objets. Il est maintenant nécessaire de prouver la validité de cette propriété.

Nous avons vu que la validité de la *TB*-activabilité reposait sur la structure arborescente du graphe des sous-systèmes et le respect des conditions de Bernstein. Cependant, nous avons également vu en 5.1.2 page 112 les limites de l'application des conditions de Bernstein dans ce contexte : la communication entre un objet actif *père* et un objet actif *fils* nécessite le partage de données entre les deux processus, que ce soit pour l'appel d'une méthode ou la transmission d'un résultat. De plus, la mobilité des objets entre sous-systèmes que nous introduisons nécessite, comme nous l'avons vu en 5.1.2 page 112, un traitement spécifique des références depuis ces objets mobiles vers un objet actif dont le sous-système serait la destination. En outre, nous introduisons la possibilité que le sous-système d'un objet actif soit composé et parcouru par plusieurs processus légers qui, de fait, partagent les mêmes données à l'intérieur de ce sous-système. Pour toutes ces raisons, la structure arborescente du graphe des sous-systèmes,[8] bien que nécessaire, ne nous paraît pas suffisante pour nous permettre d'affirmer la cohérence de cette définition de l'activabilité. En particulier, nous souhaitons prouver que l'activation de n'importe

[8] Propriété qui est conservée par notre définition.

quel sous ensemble des objets activables produit un programme dont la sémantique est similaire au programme d'origine.

La preuve que nous formulons s'appuie sur l'utilisation du π-calcul. Le principe en est le suivant : nous modélisons un objet activable par un processus π-calcul ; nous modélisons un objet actif par un autre processus π-calcul et nous prouvons la bisimulation entre ces deux processus. Ceci nous permet d'affirmer que le remplacement de l'un par l'autre dans une formule π-calcul correspondant à la traduction d'un programme Java quelconque ne change pas le comportement du processus obtenu. Différents formalismes auraient probablement pu convenir pour l'établissement de cette preuve. Nous utilisons le π-calcul pour des raisons pragmatiques : c'est un outil abondamment documenté et de façon pédagogique [93], que nous connaissons et qui nous permet de résoudre notre problème.

L'utilisation du π-calcul a nécessité la modélisation, dans ce formalisme, des objets actifs. Un travail avait déjà été effectué sur cette question par David Sagnol [111]. Cette modélisation prend en charge l'ensemble des fonctionnalités liées aux objets actifs et à la plate-forme ProActive en particulier, notamment la possibilité de définir l'ordre de traitement des appels de méthodes en attente. Dans notre cas, la définition de l'ordre des traitements n'est pas nécessaire puisque nous n'en autorisons qu'un seul : le traitement séquentiel *FIFO*, c'est-à-dire dans l'ordre de leur réception. Pour cette raison mais également afin d'obtenir un modèle le plus simple et le plus adapté possible à notre fin – prouver la validité de l'activabilité étendue – nous ne nous basons pas sur le modèle de [111] mais proposons notre propre modélisation ad-hoc.

On doit remarquer également les travaux de Denis Caromel, Ludovic Henrio et Bernard Serpette sur la définition d'un formalisme spécifique aux objets actifs [19, 68]. ASP – *Asynchronous Sequential Processes* – permet la modélisation de programmes à objets actifs ainsi que la vérification de propriétés sur ces modèles. Henrio [68] donne un certain nombre de propriétés de déterminisme pour certaines structures d'objets actifs, comme les arbres. Ces propriétés sont plus générales que celle d'activabilité définie ici dans le sens où elle permettent de spécifier des ordres de traitement des requêtes reçues par les différents objets actifs autres que la seule politique *FIFO* autorisée ici. Cependant, le formalisme ASP est spécifique aux objets actifs. En particulier, chaque processus ASP possède son propre espace mémoire, séparé de celui des autres processus ASP. Nous cherchons à modéliser et à prouver des propriétés sur des programmes ayant éventuellement recours aux processus légers. Ces processus légers partagent un même espace d'adressage, ce qui est contradictoire avec les espaces séparés des processus ASP. Il serait peut-être possible de modéliser le partage mémoire des processus légers via un mécanisme de communication évolué entre les processus ASP. Cependant, cette démarche nous semble peu naturelle, voir impossible. C'est pourquoi nous considérons que ASP ne peut pas nous permettre de prouver la validité de la propriété d'activabilité étendue.

Nous présentons maintenant la preuve de la validité de l'activabilité étendue exploitant le π-calcul. Pour cela, nous présentons d'abord le modèle π-calcul sur lequel nous nous basons pour représenter une application Java puis la modélisation des objets actifs. Nous

insisterons, dans ce modèle, sur les propriétés des objets activables et nous montrerons la bisimilarité entre un objet activable et ce même objet activé. Cela nous permettra de conclure que tout objet activable peut être remplacé par le même objet activé sans que cela n'ait d'influence observable sur la sémantique du programme.

5.3.1 Modèle π-calcul

Le modèle que nous présentons s'exprime en π-calcul. Nous donnons tout d'abord une brève définition de la syntaxe du π-calcul extraite de[93] :

Définition 5.9. *L'ensemble P^π des expressions des processus π-calcul est défini par la syntaxe suivante :*

$$P ::= \sum_{i \in I} \pi_i.P_i \;\Big|\; P_1|P_2 \;\Big|\; new\ a.P \;\Big|\; !P$$

Avec I un ensemble quelconque d'index fini et π de la forme :

$$\begin{array}{rl} \pi ::= x(y) & \textit{reçoit y via x} \\ \overline{x}\langle y\rangle & \textit{envoi y via x} \\ \tau & \textit{action inobservable} \end{array}$$

x et y sont appelés *noms*. Dans ce qui suit, nous utilisons – tout comme dans [93] – des parenthèses pour l'utilisation de noms liés et des chevrons pour l'utilisation de noms libres. Les noms de processus commencent par une majuscule.

Si S est une expression Java, nous notons $[[S]]$ la traduction de cette expression en π-calcul selon le modèle que nous définissons. Ce modèle est basé sur celui décrit dans [93].

Modélisation d'une classe Java

Soit $class\text{-}dec_C$ le code utilisé pour déclarer la classe C en Java. La traduction de ce code en π-calcul produira le processus suivant :

$$[[class\text{-}dec_C]] = !new\ c_h\big(\overline{k_C}\langle c_h\rangle.Object_C\langle c_h\rangle\big)$$

Ainsi, lorsqu'un processus écoute sur le canal public k_C, il obtient un canal restreint c_h qu'il partage avec un processus $Object_C$. Intuitivement, cette opération est équivalente a l'exécution d'une instruction `new` dans le langage Java, produisant un nouvel objet Java – le processus $Object_C$ que nous décrivons dans la suite – et l'obtention d'une référence sur cet objet – le canal c_h. Nous indiçons par h les canaux modélisant une référence Java.

Modélisation d'un objet Java

Nous décrivons maintenant la modélisation d'un objet Java, instance de la classe C et dont la référence est c_h :

$$Object_C(c_h) = !Methods_C\langle c_h\rangle \quad \text{with}$$
$$Methods_C(c_h) = new\ \overrightarrow{m}\Big(\overline{c_h}\langle\overrightarrow{m}\rangle.\sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)$$

Pour chaque méthode m_i de l'objet, $[[S_i]]$ est la traduction en π-calcul du code de cette méthode. Dans ce modèle, l'objet envoi une liste de canaux ($\overrightarrow{m}$), un par méthode pouvant être appelée, via sa référence – le canal c_h. Par la suite, une de ses méthodes peut être appelée. Par exemple, pour invoquer la i^{me} méthode de l'objet, un vecteur $\overrightarrow{p}$ de paramètres et un canal r pour l'envoi du résultat doivent être transmis via le canal m_i. Une troisième donnée – act – doit également être transmise lors d'un appel de méthode. Celle-ci nous permettra de représenter le sous-système dans lequel l'appel s'effectue. Cette notion de sous-système n'a pas de sens dans un programme Java de base et la valeur act transmise ici ne représente rien mais chaque méthode fait en sorte de retransmettre cette valeur aux méthodes qu'elle appelle. Nous verrons l'intérêt de cette donnée par la suite. Ainsi, on fait les hypothèses suivantes sur le processus $[[S_i]]$: tous les appels de méthode effectués par $[[S_i]]$ retransmettent la donnée act aux méthodes appelées et le processus se termine par le renvoi du résultat sur le canal r.

On peut également noter deux différences majeures entre ce modèle et celui proposé dans [93]. La première est le fait que les champs ne sont pas modélisés. En effet nous considérons, dans ce modèle, que l'accès aux champs d'un objet se fait toujours au travers de méthodes d'accès, nous ne modélisons pas le fonctionnement interne de ces méthodes d'accès. La seconde différence est l'absence de typage. Dans la mesure où le programme Java original traduit en π-calcul est correctement typé, nous considérons le typage de nos processus π-calcul superflu.

Illustration du modèle

Nous présentons la modélisation d'un programme Java qui crée une instance de la classe C et invoque l'une de ses méthodes. La figure 20 page suivante présente la configuration des différents processus mis en jeu.

Tout d'abord, nous créons l'instance :

$$k_C(c_h).$$

puis, au travers de la référence de la nouvelle instance créée – le canal c_h – nous récupérons la liste des méthodes appelables :

$$c_h(\overrightarrow{m}).$$

nous invoquons la i^{me} méthode de l'objet :

$$new\ r\big(\overline{m_i}\langle\overrightarrow{p},r,act\rangle.$$

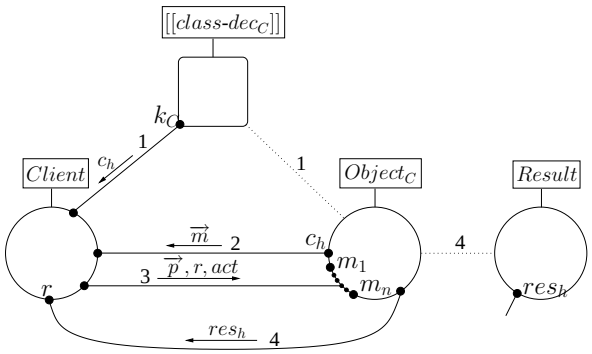


FIG. 20 – Le processus $Client$ crée une instance de la classe C avant de l'utiliser.

enfin, nous obtenons une référence sur le résultat de cette méthode :

$$r(res_h).$$

par la suite, il est possible d'utiliser ce résultat en invoquant l'une de ses méthodes, par exemple la j^{ime} :

$$res_h(\overrightarrow{m'}).new\ r'(\overline{m'_j}\langle\overrightarrow{p'}, r', act\rangle.r'(res'_h)))$$

5.3.2 Modélisation des objets actifs

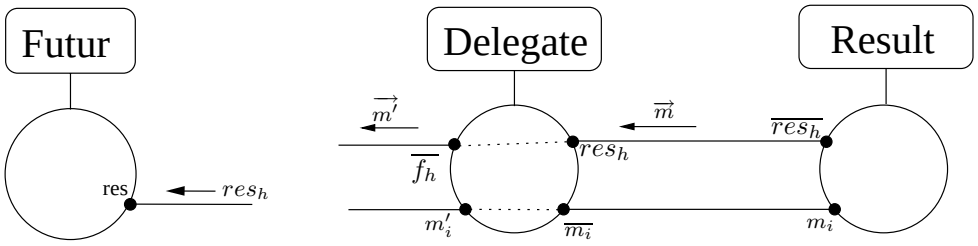
Nous allons maintenant présenter notre modélisation des objets actifs, ou plus exactement de la notion d'activation d'objet. Par rapport à cette modélisation, nous pouvons déjà donner cette définition triviale de la notion d'activabilité :

Définition 5.10. *Dans un programme Java modélisé par le processus $\mathcal{P}$, les objets activables sont ceux tels que le processus $\mathcal{P}'$ obtenu par le remplacement dans $\mathcal{P}$ de tout ou partie de ces objets par le processus $Activate$ correspondant est bisimilaire à $\mathcal{P}$.*

On pourrait traduire cette définition par *Les objets activables sont ceux tels que si on les active le programme se comporte toujours de la même façon*. Cette lapalissade ne nous permet pas encore de déduire quoi que ce soit sur l'activabilité étendue puisque la corrélation entre le processus $Activate$ et la définition de l'activabilité étendue n'est pas du tout évidente. La définition 5.11 page 126 nous permettra par la suite de faire ce rapprochement.

Modélisation des futurs

Nous commençons notre modélisation par celle des *futurs*. Lorsqu'une méthode est invoquée de façon asynchrone, le résultat de cet appel est un *futur*. Le futur représente le résultat d'un appel en cours. L'attente par nécessité bloque celui qui tente d'utiliser un futur jusqu'à ce que l'appel dont il représente le résultat soit réellement terminé. Lorsque



(a) Avant que le résultat ne soit disponible

(b) Une fois le résultat disponible

FIG. 21 – Le processus $Future$

ce résultat est disponible, le rôle du futur est simplement de transmettre tous les appels de méthode reçus vers le véritable objet. En voici notre modélisation :

$$\begin{aligned} Future(res, f_h) &= res(res_h).!Delegate\langle res_h, f_h\rangle \\ Delegate(res_h, f_h) &= res_h(\overrightarrow{m}). \\ &\quad new\ \overrightarrow{m'}\Big(\overline{f_h}\langle\overrightarrow{m'}\rangle.\sum_i\Big\{m'_i(\overrightarrow{p}, r, act). \\ &\qquad new\ r'\big(\overline{m_i}\langle\overrightarrow{p}, r', act\rangle.r'(res_h).\overline{r}\langle res_h\rangle\big)\Big\}\Big) \end{aligned}$$

Ce processus $Future$ reçoit deux paramètres. Le premier, res, est un canal sur lequel il recevra le véritable résultat de la méthode lorsqu'il sera disponible. Le second, f_h, est une référence sur lui-même, c'est-à-dire le canal qui sera utilisé pour contacter le futur et invoquer les méthodes du résultat à travers ce dernier.

Le processus $Delegate$ est gardé par le canal res ainsi le futur est inutilisable par l'appelant tant que celui-ci n'a rien reçu depuis ce canal res, c'est-à-dire, comme nous le verrons par la suite, tant que la méthode asynchrone n'est pas terminée et n'a pas retourné le véritable résultat. La donnée reçue sur ce canal, res_h, est la référence sur le véritable résultat de la méthode. Une fois cette donnée reçue, le processus $Delegate$ peut démarrer. Il reçoit les appels de méthode vers le résultat via le canal f_h et les transmet via res_h. On notera que $\overrightarrow{m'}$ et $\overrightarrow{m}$ sont de la même taille. La figure 21 représente les configurations des processus avant et après la récupération de res_h via res.

Modélisation d'un objet actif

Nous modélisons maintenant un objet actif en π-calcul en exploitant le processus $Future$ que nous venons de définir. Pour rappel : un objet actif est associé à un sous-système, les appels internes au sous-système sont directs, les appels externes sont asynchrones et retournent un futur ; ils sont traités dans l'ordre *FIFO* de leur arrivée.

Le principe de l'implantation d'un tel système est le suivant : un *proxy* simule l'objet activé. Les appels internes au sous-système associé effectués sur ce proxy sont directement délégués à l'objet sous-jacent et peuvent donc s'exécuter éventuellement de façon

concurrente. Les appels externes sont, par contre, traités séquentiellement (utilisation du canal $done$ dans ce qui suit) et stockés dans une file puis un futur est retourné à l'appelant. Un processus est également associé à l'objet. Ce processus lit les appels stockés dans la file un par un et effectue ces appels sur le véritable objet de façon séquentielle. Lorsque la véritable méthode termine, le futur correspondant à l'appel est mis à jour par ce processus puis un autre appel en attente dans la file est traité.

Nous proposons l'expression suivante pour le processus $Activate_l(o_h, oa_h, act_l)$. Ce processus active l'objet référencé par le canal o_h. L'objet activé est ensuite accessible via le canal oa_h. L'indice l et le canal act_l représentent l'identité du sous-système associé :

$$
\begin{aligned}
&Activate_l(o_h, oa_h, act_l) = new\ queue, unqueue, newCell, exec, done\Big(\\
&\quad !Cell|!Exec| \\
&\quad \overline{newCell}\langle unqueue\rangle|\overline{exec}\langle unqueue\rangle|\overline{done}| \\
&\quad !new\ \overrightarrow{m'}, \overrightarrow{a}\Big(o_h(\overrightarrow{m}).\overline{oa_h}\langle\overrightarrow{m'}\rangle. \\
&\qquad \sum_i \Big\{m'_i(\overrightarrow{p}, r, act).\overline{act}\langle\overrightarrow{a}\rangle. \\
&\qquad\quad \big[a_l.Call + \\
&\qquad\qquad \sum_{j=0..l-1,l+1..A} \{a_j.done.new\ res, f_h(Future\langle res, f_h\rangle|ActivCall.\overline{done})\}\big]\Big\}\Big)\Big)
\end{aligned}
$$

Le sous-système correspondant à ce processus a pour identité l et cette identité est modélisée par le canal act_l qui, comme nous le verrons par la suite, sera transmis pour tous les appels de méthodes effectués par ce processus. Lorsqu'un appel est reçu sur un canal m_i, l'identité du sous-système appelant est représentée par le canal act transmis lors de l'appel. Afin de savoir si l'appel est interne ou externe au sous-système, le processus transmet un vecteur $\overrightarrow{a}$ de canaux supposé de taille A – le nombre maximum de sous-systèmes dans le programme – puis attend une réponse sur un de ces canaux. Si la réponse est reçue sur le $l^{\text{ième}}$ canal alors l'appel est interne et donc direct ($a_l.Call$) sinon, si une réponse est reçue sur n'importe quel autre canal du vecteur alors l'appel est externe ($\sum_{j=0..l-1,l+1..A}\{a_j.\ldots$). Pour permettre l'identification des sous-systèmes, un processus $SubSystem_i$ existe pour chaque sous-système i. Son rôle est simplement de répondre sur le $i^{\text{ième}}$ canal du vecteur qui lui est transmis :

$$SubSystem_i = act_i(\overrightarrow{a}).\overline{a_i}$$

Chaque objet actif possède son propre sous-système et est donc associé à son propre processus $SubSystem$. De plus, on considère qu'il existe également un sous-système par défaut auquel est associé le programme principal.

Le processus $Call$ est exécuté dans le cas où l'appel est effectué depuis l'intérieur du sous-système – réponse reçue sur a_l. Dans ce cas, l'appel est direct et plusieurs appels peuvent se dérouler simultanément s'ils sont concurrents :

$$Call = new\ r'(\overline{m_i}\langle\overrightarrow{p}, r', act_l\rangle.r'(res_h).\overline{r}\langle res_h\rangle)$$

Dans le cas où l'appel est externe, c'est-à-dire que l'appelant est dans un sous-système distinct, alors il sera traité séquentiellement vis-à-vis des autres appels externes. Pour cela, chaque processus est gardé par une communication sur le canal $done$ puis débloque le suivant ($\overline{done}$) à la fin de son exécution. On peut d'ailleurs noter la présence d'un processus isolé n'effectuant que $\overline{done}$ sur la troisième ligne de la formule de $Activate_l$ et dont le seul rôle est de débloquer le premier processus attendant sur $done$. Le traitement d'un appel externe consistera ensuite à placer cet appel dans la file d'appels et à retourner un futur à l'appelant. C'est le processus $ActivCall$ qui a pour rôle de placer l'appel dans la file et de retourner une référence sur le futur correspondant :

$$ActivCall = \overline{queue}\langle(m_i, \overrightarrow{p}, res)\rangle.\overline{r}\langle f_h\rangle$$

!$Cell$ modélise la file des appels. Une nouvelle cellule dans la file est *créée* par une communication sur le canal $newCell$. Un canal permettant de lire le contenu de cette cellule lui est transmis via $newCell$:

$$Cell = \ newCell(u).queue(elt).\ new\ u'(\overline{newCell}\langle u'\rangle|\overline{u}\langle elt, u'\rangle)$$

Une fois la cellule créée, on constate qu'un élément peut y être placé via le canal non lié $queue$. Une nouvelle cellule est alors créé dont le contenu pourra être lu via u' : $\overline{newCell}\langle u'\rangle$. Parallèlement à cela, la donnée reçue sur $queue$ ainsi que le canal de lecture de la nouvelle cellule sont rendus disponibles sur u : $\overline{u}\langle elt, u'\rangle$. On peut noter que $Activate_l$ transmet un message sur le canal $newCell$ afin d'initialiser une première cellule. $Activate_l$ fournit également le canal $unqueue$ permettant de lire la donnée contenue dans cette première cellule au processus $Exec$.

!$Exec$ modélise le processus chargé d'effectuer les appels de méthode stockés dans la file. Ce processus lit les appels à effectuer dans la file. $Exec$ est gardé par le canal $exec$ sur lequel il reçoit le canal u permettant de lire depuis la prochaine case de la file l'appel à effectuer. Cet appel est décrit par le canal de la méthode à invoquer, la liste des arguments ainsi que le canal sur lequel transmettre le résultat de l'appel au futur correspondant. Lorsque l'appel est effectué, il transmet le résultat au futur puis permet à un autre processus $Exec$ de traiter l'élément suivant dans la file ($\overline{exec}\langle u'\rangle$). D'autre part, lorsque $Exec$ invoque une méthode, il transmet, pour sous-système d'origine, act_l, c'est-à-dire le sous-système correspondant à cet objet actif :

$$Exec = exec(u).u((m_i, \overrightarrow{p}, res), u').new\ r'(\overline{m_i}\langle\overrightarrow{p}, r', act_l\rangle.r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u'\rangle)$$

On peut noter que le canal res, permettant la mise à jour du futur, aurait pu directement être transmis, lors de l'appel de la méthode, comme canal de retour de résultat. De cette façon, le futur aurait directement reçu le résultat de la méthode via res. Mais dans ce cas le processus $Exec$ ne serait plus bloqué le temps que la méthode soit terminée et cela pourrait provoquer l'appel en parallèle de plusieurs méthodes. Cette solution est donc à proscrire, notre processus attend explicitement le résultat de la méthode avant de le transmettre lui-même au futur.

5.3.3 Modélisation d'un objet activable

L'objectif de cette modélisation en π-calcul est de prouver qu'un objet respectant le critère d'activabilité étendue de la définition 5.8 page 118 peut être activé sans que le comportement du programme n'en soit altéré. Pour cela, il nous est nécessaire de formuler un critère d'activabilité reposant sur le π-calcul et dont on puisse se convaincre qu'il est bien en relation avec la définition 5.8 page 118. Le premier critère π-calcul de la définition 5.10 page 122 ne nous aide pas sur ce point, nous donnons donc une nouvelle définition en terme π-calcul de la notion d'activabilité :

Définition 5.11. *Dans un programme Java modélisé par le processus $\mathcal{P}$, les objets activables sont ceux tels que le processus $\mathcal{P}'$ obtenu par le remplacement dans $\mathcal{P}$ de tout ou partie de ces objets par le processus $Activable$ correspondant est bisimilaire à $\mathcal{P}$.*

Nous allons maintenant présenter le processus $Activable$ de cette définition. L'objectif est de nous permettre de nous convaincre, de façon informelle, que le processus modélisant un objet activable au sens de la définition 5.8 page 118 est bisimilaire au processus $Activable$ correspondant. Pour cela nous allons intégrer dans ce processus les éléments de la définition 5.8 page 118. Nous prouverons par la suite que, pour un objet donné, le processus $Activable$ est bisimilaire au processus $Activate$.

La modélisation d'un objet activable ne diffère à priori pas de celle d'un objet standard, pour rappel :

$$Object_C(c_h) = !Methods_C\langle c_h\rangle \quad \text{avec}$$
$$Methods_C(c_h) = new\ \overrightarrow{m}\Big(\overline{c_h}\langle\overrightarrow{m}\rangle.\sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)$$

Cependant, selon la propriété d'activabilité étendue que nous avons définie, deux caractéristiques différencient les objets activables des objets normaux.

La première est le fait que tous les chemins de l'extérieur vers l'intérieur de l'accessibilité d'un objet activable passent par cet objet activable. Cela signifie que tous les objets appartenant au sous-système d'un objet activable ne peuvent être utilisés que de l'intérieur de ce sous-système. Autrement dit, toutes les opérations effectuées par les méthodes d'un objet activable sont invisibles pour un appelant extérieur à ce sous-système, mis à part l'envoi du résultat lui-même. Ainsi, lorsqu'une méthode m_i est invoquée de l'extérieur du sous-système, le terme $[[S_i]]$ de cette méthode se réduit par une suite de τ-transitions en $\overline{r}\langle res_h\rangle$, res_h étant la référence sur le résultat de la méthode, c'est-à-dire simplement le retour du résultat à l'appelant. De plus, tout résultat retourné à l'extérieur de son sous-système par une de ses méthodes n'est plus référencé ou *potentiellement utilisable* (Cf. définition 5.6 page 116) depuis l'intérieur de son sous-système après cet envoi. Autrement dit, tout objet transmis comme résultat d'une méthode lors d'un appel depuis l'extérieur du sous-système quitte le sous-système. On peut donc considérer le résultat comme accessible uniquement par l'appelant après le retour de la méthode.

Finalement, on peut en déduire que, lors d'un appel externe, tous les termes $[[S_i]]$ d'un objet activable peuvent se réduire, par une suite de τ-transitions en :

$$new\ res_h(Object\langle res_h\rangle|\overline{r}\langle res_h\rangle)$$

La seconde particularité des objets activables est le fait que les méthodes de ces objets ne sont invoquées, de l'extérieur de leur sous-système, que par un seul processus léger *à la fois*. Ces appels externes peuvent donc être séquentialisés.

Finalement, nous proposons le processus π-calcul suivant pour modéliser un objet activable :

$$\begin{aligned}
Activable_l(o_h, act_l) = new\ c_h\Big(&Object_C\langle c_h\rangle\Big| \\
&new\ done.\Big\{\overline{done}\Big| \\
&\quad !\Big[c_h(\overrightarrow{m}).new\ \overrightarrow{m'}.\overline{o_h}\langle\overrightarrow{m'}\rangle. \\
&\quad\quad \sum_i\Big\{m'_i(\overrightarrow{p}, r, act).new\ \overrightarrow{a}.\overline{act}\langle\overrightarrow{a}\rangle. \\
&\quad\quad\quad [a_l.Call + \sum_{j=0..l-1,l+1..A} a_j.done.Call.\overline{done}]\Big\}\Big]\Big\}\Big)
\end{aligned}$$

Nous exploiterons plus tard la réduction des termes $[[S_i]]$. Cependant, on peut déjà constater que ce processus modélise un objet dont les appels internes à son sous-système sont effectués de façon standard mais les appels externes de façon séquentielle. Ainsi, on peut se convaincre que la définition 5.11 page précédente correspond bien à l'activabilité étendue.

Nous avons déjà publié une première version de la preuve de la validité de l'activabilité étendue exploitant le π-calcul [25]. Bien que les problèmes de récursivité aient été mis en évidence dans cet article, les problèmes de récursivité indirecte tels que posés par la mobilité d'objets n'avaient pas été relevés et cela pouvait entraîner des étreintes fatales. Ce nouveau modèle résout ce problème puisqu'on voit bien que le processus $Activable$ pose explicitement le fait que les appels doivent être asynchrones séquentiels depuis l'extérieur et directs lorsqu'ils sont internes.

5.3.4 Bisimulation

Si nous considérons que la définition 5.11 page ci-contre correspond bien à la définition de l'activabilité étendue, prouver que l'activation d'un objet activable n'altère pas le comportement du programme consiste à prouver que le processus modélisant un objet activable est en bisimulation avec le processus modélisant ce même objet activé.

Posons d'abord P, l'objet activable, accessible via le canal o_h et dont le sous-système

a l'identité act_l :

$$
\begin{aligned}
P = new\ c_h, done\Big(& \\
& Object_C\langle c_h\rangle \Big| \overline{done} \Big| \\
& !new\ \overrightarrow{m'}, \overrightarrow{a}\Big(c_h(\overrightarrow{m}).\overline{o_h}\langle\overrightarrow{m'}\rangle. \\
& \qquad \sum_i \Big\{ m'_i(\overrightarrow{p}, r, act).\overline{act}\langle\overrightarrow{a}\rangle. \\
& \qquad\qquad [a_l.Call + \sum_{j=0..l-1,l+1..A} \Big\{ a_j.done.Call.\overline{done} \Big\}] \Big\} \Big) \Big)
\end{aligned}
$$

Posons maintenant Q, le même objet activé et accessible également via le canal o_h :

$$
\begin{aligned}
Q = new\ c_h, done, queue, unqueue, newCell, exec\Big(& \\
& Object_C\langle c_h\rangle \Big| \overline{done} \Big| \\
& !Cell \Big| !Exec \Big| \\
& \overline{newCell}\langle unqueue\rangle \Big| \overline{exec}\langle unqueue\rangle \Big| \\
& !new\ \overrightarrow{m'}, \overrightarrow{a}\Big(c_h(\overrightarrow{m}).\overline{o_h}\langle\overrightarrow{m'}\rangle. \\
& \qquad \sum_i \Big\{ m'_i(\overrightarrow{p}, r, act).\overline{act}\langle\overrightarrow{a}\rangle. \\
& \qquad\qquad [a_l.Call + \\
& \qquad\qquad\qquad \sum_{j=0..l-1,l+1..A} \Big\{ a_j.done.new\ res, f_h(Future\langle res, f_h\rangle | ActivCall.\overline{done}) \Big\}] \Big\} \Big) \Big)
\end{aligned}
$$

Rappels

Avant d'aller plus loin nous effectuons quelques brefs rappels au sujet du π-calcul. Ces éléments sont issus de [93] auquel on se référera pour plus de détails. Dans la suite, on notera :

1. $P \xrightarrow{\lambda} P'$ lorsque le processus P peut se transformer en P' par une transition λ ;
2. $P \longrightarrow P'$ lorsque le processus P peut se transformer en P' par une τ-transition ;
3. $P \stackrel{\lambda}{\Longrightarrow} P'$ lorsque le processus P peut se transformer en P' par une suite de 0 ou n τ-transitions suivi d'une transition λ suivi de 0 ou n τ-transitions ;
4. $fn(P)$ l'ensemble des variables libres du processus P.

Congruence structurelle La congruence structurelle, notée $\equiv$, est déterminée par :

1. Le changement des noms liés

2. La modification de l'ordre des termes dans une somme
3. $P|0 \equiv P$, $P|Q \equiv Q|P$, $P|(Q|R) \equiv (P|Q)|R$
4. $new\ a\ (P|Q) \equiv P|new\ a\ Q$ si $a \notin fn(P)$
   $new\ a\ 0 \equiv 0$, $new\ a, b\ P \equiv new\ b, a\ P$
5. $A\langle \overrightarrow{b} \rangle \equiv \{^{\overrightarrow{b}}/_{\overrightarrow{a}}\}P_A$ si $A(\overrightarrow{a}) = P_A$

Bisimulation forte Une relation binaire S est une simulation forte si, chaque fois que $(P, Q) \in S$ (noté PSQ), si $P \xrightarrow{\alpha} P'$ alors il existe Q' tel que $Q \xrightarrow{\alpha} Q'$ et $P'SQ'$.

S est une bisimulation forte (noté $\sim$) si sa relation inverse respecte également cette propriété.

Bisimulation faible Une relation binaire S est une simulation faible si, chaque fois que PSQ,

1. si $P \longrightarrow P'$ alors il existe Q' tel que $Q \Longrightarrow Q'$ et $P'SQ'$;
2. si $P \xrightarrow{\lambda} P'$ alors il existe Q' tel que $Q \xRightarrow{\lambda} Q'$ et $P'SQ'$.

S est une bisimulation faible (noté $\approx$) si sa relation inverse respecte également cette propriété.

Simplification de P

$$P \equiv new\ done\Big(\overline{done}\Big|new\ c_h(!Methods_C\langle c_h\rangle|!c_h(\overrightarrow{m}).P')\Big) \quad \text{avec}$$
$$P' = new\ \overrightarrow{m'}, \overrightarrow{a}\ \Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i\Big\{m'_i(\overrightarrow{p}, r, act).\overline{act}\langle\overrightarrow{a}\rangle.$$
$$[a_l.Call + \sum_{j=0..l-1,l+1..A}\Big\{a_j.done.Call.\overline{done}\Big\}]\Big\}\Big)$$

On dit d'un processus qu'il est négatif pour x si les seules occurrences de x dans ce processus sont de la forme $\overline{x}.C$. Ainsi, $Methods_C\langle c_h\rangle$ et P' sont négatifs pour c_h. Nous pouvons donc appliquer le théorème 12.36 de [93] :

Théorème. *Soient P et F négatifs pour x. Alors*

$$new\ x(!P|!xF) \sim !new\ x(P|!xF)$$

On en déduit :

$$P \sim new\ done\Big(\overline{done}\Big|!new\ c_h(Methods_C\langle c_h\rangle|!c_h(\overrightarrow{m}).P')\Big)$$

De plus, $Methods_C\langle c_h\rangle$ n'utilise c_h qu'une seule fois. La réplication de $c_h(\overrightarrow{m}).P'$ est donc superflue :

$$P \sim new\ done\Big(\overline{done}\Big|!new\ c_h(Methods_C\langle c_h\rangle|c_h(\overrightarrow{m}).P')\Big)$$

L'expression $!new\ c_h(Methods_C\langle c_h\rangle|c_h(\overrightarrow{m}).P')$ ne peut évoluer que par une τ-transition sur c_h. Ainsi, selon la loi d'expansion (*expansion law*, proposition 5.23 de [93]) :

$$\begin{aligned}P \sim new\ done\Big(\overline{done}\Big| \\ !\tau.new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\Big[\sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big| \\ \sum_i\Big\{m'_i(\overrightarrow{p},r,act).\overline{act}\langle\overrightarrow{a}\rangle. \\ [a_l.Call+\sum_{j=0..l-1,l+1..A}\Big\{a_j.done.Call.\overline{done}\Big\}]\Big\}\Big]\Big)\Big)\end{aligned}$$

Lemme.

$$new\ \overrightarrow{x}\Big(\sum_i\{a_i.F_i\}\Big|\sum_j\{x_j.G_j\}\Big)\sim\sum_i\Big\{a_i.new\overrightarrow{x}\Big(F_i\Big|\sum_j\{x_j.G_j\}\Big)\Big\}$$

Ainsi, et par expansion sur act :

$$\begin{aligned}P \sim new\ done\Big(\overline{done}\Big| \\ !\tau.new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i\Big\{m'_i(\overrightarrow{p},r,act).\overline{act}\langle\overrightarrow{a}\rangle. \\ \Big[\sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big| \\ [a_l.Call+\sum_{j=0..l-1,l+1..A}\Big\{a_j.done.Call.\overline{done}\Big\}]\Big]\Big\}\Big)\Big)\end{aligned}$$

De même :

$$\begin{aligned}P \sim new\ done\Big(\overline{done}\Big| \\ !\tau.new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i\Big\{m'_i(\overrightarrow{p},r,act).\overline{act}\langle\overrightarrow{a}\rangle. \\ \Big[a_l.\Big(Call|\sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)+ \\ \sum_{j=0..l-1,l+1..A}\Big\{a_j.\Big(done.Call.\overline{done}| \\ \sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)\Big\}\Big]\Big\}\Big)\Big)\end{aligned}$$

Par expansion des τ-transitions sur m_i :

$$P \sim new\ done\Big(\overline{done} \Big| !\tau.new\ \overrightarrow{m}, \overrightarrow{m'}, \overrightarrow{a}\Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i \Big\{ m'_i(\overrightarrow{p}, r, act).\overline{act}\langle\overrightarrow{a}\rangle. \Big[a_l.\tau.new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle \Big| \{^{\overrightarrow{p}}/_{\overrightarrow{p}}\}\{^{r'}/_{r}\}\{^{act}/_{act_l}\}[[S_i]]\Big) + \sum_{j=0..l-1,l+1..A} \Big\{ a_j.done.\tau.new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle.\overline{done} \Big| \{^{\overrightarrow{p}}/_{\overrightarrow{p}}\}\{^{r'}/_{r}\}\{^{act}/_{act_l}\}[[S_i]]\Big)\Big\}\Big]\Big\}\Big)\Big)$$

Comme nous l'avons vu précédemment, dans le cas où un appel de méthode est effectué depuis l'extérieur, les termes $[[S_i]]$ d'un objet activable se dérivent par τ-transitions en :

$$new\ res_h(Object\langle res_h\rangle | \overline{r}\langle res_h\rangle)$$

Dans notre cas, c'est le contexte du deuxième terme $[[S_i]]$ qui correspond à un appel externe :

$$P \approx new\ done\Big(\overline{done} \Big| !\tau.new\ \overrightarrow{m}, \overrightarrow{m'}, \overrightarrow{a}\Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i \Big\{ m'_i(\overrightarrow{p}, r, act).\overline{act}\langle\overrightarrow{a}\rangle. \Big[a_l.\tau.new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle \Big| \{^{\overrightarrow{p}}/_{\overrightarrow{p}}\}\{^{r'}/_{r}\}\{^{act}/_{act_l}\}[[S_i]]\Big) + \sum_{j=0..l-1,l+1..A} \Big\{ a_j.done.\tau.new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle.\overline{done} \Big| new\ res_h(Object\langle res_h\rangle | \overline{r'}\langle res_h\rangle)\Big)\Big\}\Big]\Big\}\Big)\Big)$$

Finalement :

$P \approx \mathbb{P}$ avec

$$\mathbb{P} = new\ done\Big(\overline{done} \Big| !new\ \overrightarrow{m}, \overrightarrow{m'}, \overrightarrow{a}\Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i \Big\{ m'_i(\overrightarrow{p}, r, act).\overline{act}\langle\overrightarrow{a}\rangle. \Big[a_l.new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle \Big| \{^{\overrightarrow{p}}/_{\overrightarrow{p}}\}\{^{r'}/_{r}\}\{^{act}/_{act_l}\}[[S_i]]\Big) + \sum_{j=0..l-1,l+1..A} \Big\{ a_j.done.new\ res_h\Big(\overline{r}\langle res_h\rangle.\overline{done} | Object\langle res_h\rangle\Big)\Big\}\Big]\Big\}\Big)\Big)$$

Simplification de Q

De la même façon :

$$
\begin{aligned}
&Q \approx \mathbb{Q} \quad \text{avec}\\
&\mathbb{Q} = new\ done, queue, unqueue, newCell, exec\Big(\\
&\quad \overline{done}\Big|\overline{newCell}\langle unqueue\rangle\Big|\overline{exec}\langle unqueue\rangle\Big|\\
&\quad !Cell\Big|!Exec\Big|\\
&\quad !new\ \overrightarrow{m}, \overrightarrow{m'}, \overrightarrow{a}\Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i\Big\{m'_i(\overrightarrow{p}, r, act).\overline{act}\langle\overrightarrow{a}\rangle.\\
&\qquad\qquad \Big[a_l.new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle\Big|\{^{\overrightarrow{p}}/_{\overrightarrow{p}}\}\{^{r'}/_{r}\}\{^{act}/_{act_l}\}[[S_i]]\Big)+\\
&\qquad\qquad\qquad \sum_{j=0..l-1,l+1..A}\Big\{a_j.done.new\ res, f_h\Big(Future\langle res, f_h\rangle|\\
&\qquad\qquad\qquad\qquad \sum_i\Big\{m_i(\overrightarrow{p}, r, act).[[S_i]]\Big\}|\\
&\qquad\qquad\qquad\qquad ActivCall.\overline{done}\Big)\Big\}\Big]\Big\}\Big)\Big)
\end{aligned}
$$

Bisimulation entre $\mathbb{P}$ et $\mathbb{Q}$

Afin de prouver la bisimulation entre $\mathbb{P}$ et $\mathbb{Q}$, nous construisons une relation $\mathcal{S}$ qui est une bisimulation et dont $\mathbb{P}$ et $\mathbb{Q}$ font partie. Afin de simplifier les formules nous posons d'abord $\mathbb{P}'(a, b, c, d, e, f, \overrightarrow{x}, \overrightarrow{y})$ et $\mathbb{Q}'(a, b, c, d, e, f, g, h, \overrightarrow{x}, \overrightarrow{l}, \overrightarrow{m}, \overrightarrow{n})$:

Dans l'équation qui suit, on remplace : $\Big[a_l.new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle\Big|\{^{\overrightarrow{p}}/_{\overrightarrow{p}}\}\{^{r'}/_{r}\}\{^{act}/_{act_l}\}[[S_i]]\Big)+ \sum_{j=0..l-1,l+1..A}\Big\{a_j.done.new\ res_h\Big(\overline{r}\langle res_h\rangle.\overline{done}\Big|Object\langle res_h\rangle\Big)\Big\}\Big]$

par : $\Big[a_l \ldots + \sum_{j=0..l-1,l+1..A}\Big\{a_j \ldots\Big\}\Big]$

$$\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y}) =$$

$$!new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i\Big\{m'_i(\overrightarrow{p},r,act).\overline{act}\langle\overrightarrow{a}\rangle.\Big[a_l\ldots+\sum_{j=0..l-1,l+1..A}\Big\{a_j\ldots\Big\}\Big]\Big\}\Big)\Big|$$

$$\prod_a new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\Big(\sum_i\Big\{m'_i(\overrightarrow{p},r,act).\overline{act}\langle\overrightarrow{a}\rangle.\Big[a_l\ldots+\sum_{j=0..l-1,l+1..A}\Big\{a_j\ldots\Big\}\Big]\Big\}\Big)\Big|$$

$$\prod_b new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\Big(\overline{act}\langle\overrightarrow{a}\rangle.\Big[a_l\ldots+\sum_{j=0..l-1,l+1..A}\Big\{a_j\ldots\Big\}\Big]\Big)\Big|$$

$$\prod_c new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle.\Big[a_l\ldots+\sum_{j=0..l-1,l+1..A}\Big\{a_j\ldots\Big\}\Big]\Big|$$

$$\prod_d new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle\Big|\{\overrightarrow{p}/_{\overrightarrow{p}}\}\{r'/_r\}\{act/_{act_l}\}[[S_i]]\Big)\Big|$$

$$\prod_e new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\Big(done.new\ res_h\Big(\overline{r}\langle res_h\rangle.\overline{done}\Big|Object\langle res_h\rangle\Big)\Big)\Big|$$

$$\prod_{f'=1..f} new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \langle res_h\rangle\Big(Object\langle res_h\rangle\Big|$$

$$\prod_{x_{f'}} new\ \overrightarrow{m''}\ \langle\overrightarrow{m''}\rangle\sum_i\Big\{m''_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big|$$

$$\prod_{y_{f'}} new\ \overrightarrow{m''}\ \langle\overrightarrow{m''}\rangle\ (\overrightarrow{p},r,act)[[S_i]]\Big)$$

Dans l'équation qui suit, on remplace : $\Big[a_l.new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle\Big|\{\overrightarrow{p}/_{\overrightarrow{p}}\}\{r'/_r\}\{act/_{act_l}\}[[S_i]]\Big)+$

$$\sum_{j=0..l-1,l+1..A}\Big\{a_j.done.new\ res,f_h\Big(Future\langle res,f_h\rangle\Big|$$

$$\sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big|$$

$$ActivCall.\overline{done}\Big)\Big\}\Big]$$

par : $\Big[a_l\ldots+\sum_{j=0..l-1,l+1..A}\Big\{a_j\ldots\Big\}\Big]$

$$
\begin{aligned}
&\mathbb{Q}'(a,b,c,d,e,f,g,h,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n}) = \\
&\quad !Cell \Big| !Exec \Big| \\
&\quad !new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\Big(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i\Big\{m'_i(\overrightarrow{p},r,act).\overline{act}\langle\overrightarrow{a}\rangle.\Big[a_l\ldots+\sum_{j=0..l-1,l+1..A}\Big\{a_j\ldots\Big\}\Big]\Big\}\Big)\Big| \\
&\quad \prod_a new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\Big(\sum_i\Big\{m'_i(\overrightarrow{p},r,act).\overline{act}\langle\overrightarrow{a}\rangle.\Big[a_l\ldots+\sum_{j=0..l-1,l+1..A}\Big\{a_j\ldots\Big\}\Big]\Big\}\Big)\Big| \\
&\quad \prod_b new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\Big(\overline{act}\langle\overrightarrow{a}\rangle.\Big[a_l\ldots+\sum_{j=0..l-1,l+1..A}\Big\{a_j\ldots\Big\}\Big]\Big)\Big| \\
&\quad \prod_c new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle.\Big[a_l\ldots+\sum_{j=0..l-1,l+1..A}\Big\{a_j\ldots\Big\}\Big]\Big| \\
&\quad \prod_d new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\ new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle\Big|\{\overrightarrow{p}/_{\overrightarrow{p}}\}\{r'/_r\}\{act/_{act_l}\}[[S_i]]\Big)\Big| \\
&\quad \prod_e new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(done.new\ res, f_h\Big(Future\langle res,f_h\rangle\Big|\sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big| \\
&\qquad\qquad ActivCall.\overline{done}\Big)\Big) \\
&\quad \prod_{f'=g+1..f-h} new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big|\sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big| \\
&\qquad\qquad \overline{u_{f'}}\langle(m_i,\overrightarrow{p},res),u_{f'+1}\rangle\Big)\Big| \\
&\quad \prod_{g'=1..g} new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big\{!Delegate\langle res_h,f_h\rangle\Big|Object\langle res_h\rangle\Big| \\
&\qquad \prod_{l_{g'}} new\ \overrightarrow{m},\overrightarrow{m'}\Big(\overline{f_h}\langle\overrightarrow{m'}\rangle.\sum_i\Big\{{m'}_i(\overrightarrow{p},r,act).new\ r'(\overline{m_i}\langle\overrightarrow{p},r',act\rangle.r'(res_h).\overline{r}\langle res_h\rangle)\Big\}\Big| \\
&\qquad\qquad \sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)\Big| \\
&\qquad \prod_{x_{g'}} new\ \overrightarrow{m},\overrightarrow{m'}\ \langle\overrightarrow{m'}\rangle\Big(\sum_i\Big\{{m'}_i(\overrightarrow{p},r,act).new\ r'(\overline{m_i}\langle\overrightarrow{p},r',act\rangle.r'(res_h).\overline{r}\langle res_h\rangle)\Big\}\Big| \\
&\qquad\qquad \sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)\Big| \\
&\qquad \prod_{m_{g'}} new\ \overrightarrow{m},\overrightarrow{m'}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ new\ r'\Big(\overline{m_i}\langle\overrightarrow{p},r',act\rangle.r'(res_h).\overline{r}\langle res_h\rangle\Big| \\
&\qquad\qquad \sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)\Big| \\
&\qquad \prod_{n_{g'}} new\ \overrightarrow{m},\overrightarrow{m'}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle\Big| \\
&\qquad\qquad \{\overrightarrow{p}/_{\overrightarrow{p}}\}\{r'/_r\}\{act/_{act}\}[[S_i]]\Big)\Big\}
\end{aligned}
$$

Ainsi, on peut écrire $\mathbb{P}$ et $\mathbb{Q}$ en fonction de $\mathbb{P}'$ et $\mathbb{Q}'$:

$$\mathbb{P} = new\ done\Big(\overline{done}\Big|\mathbb{P}'(0,0,0,0,0,0,\emptyset,\emptyset)\Big)$$

$$\mathbb{Q} = new\ done, newCell, exec, q, u_1\Big(\overline{done}\Big|\overline{newCell}\langle u_1\rangle\Big|\overline{exec}\langle u_1\rangle\Big|\mathbb{Q}'(0,0,0,0,0,0,0,0,\emptyset,\emptyset,\emptyset,\emptyset)\Big)$$

Les deux expressions $\mathbb{P}'$ et $\mathbb{Q}'$ ont recours à la notion d'agent (définition 12.1 de [93]). Un agent correspond au *résidu* d'un processus après une réaction. Par exemple le processus :

$$new\ \overrightarrow{m'}\left(\overline{o_h}\langle\overrightarrow{m'}\rangle.\sum_i\Big\{m'_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\right)$$

Se transforme, par réaction sur o_h en l'agent :

$$new\ \overrightarrow{m'}\langle\overrightarrow{m'}\rangle\left(\sum_i\Big\{m'_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\right)$$

De même le processus :

$$\sum_i\Big\{m'_i(\overrightarrow{p},r,act).[[S_i]]\Big\}$$

Se transforme, par réaction sur l'un des m'_i en :

$$(\overrightarrow{p},r,act)[[S_i]]$$

Dans l'expression $\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})$, les rôles des différentes valeurs sont :

- a correspond au nombre de processus ayant réagit par o_h et qui sont prêts à recevoir un appel de méthode via l'un des canaux privés m'_i transmis. Ces agents peuvent se combiner avec d'autres agents résidus d'une *lecture* sur o_h, c'est-à-dire avec les utilisateurs de l'objet en question.
- b correspond au nombre de processus ayant réagi par o_h puis ayant reçu un appel sur un canal privé m'_i.
- c correspond au nombre de processus ayant effectué une réaction supplémentaire.
- d est le nombre de processus dont l'appel de méthode est interne et qui ont effectivement transmis cet appel à l'objet sous-jacent.
- e est le nombre de processus dont l'appel est externe et qui attendent de pouvoir l'effectuer (traitement séquentiel des appels externes).
- f correspond au nombre d'appels externes effectués ; les processus correspondant sont les résultats de ces appels.
- $x_{f'}$ et $y_{f'}$ représentent le nombre d'appels en cours sur un résultat f'.

Dans l'expression $\mathbb{Q}'(a,b,c,d,e,f,g,h,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})$, les valeurs a, b, c, d, e, f et $\overrightarrow{x}$ ont la même signification que précédemment et :

- g correspond au nombre de résultats effectivement disponibles alors que $f-g$ correspond au nombre de résultats retournés (futurs) non encore disponibles.

– h est une variable d'ajustement, elle vaut 0 ou 1 selon l'expression dans laquelle elle est utilisée. On comprendra son utilisation dans les formules de la relation $\mathcal{S}$.

– $l_{g'}$ correspond au nombre de processus ayant effectué une réaction interne sur le canal privé res_h d'un résultat g'.

– $m_{g'} + n_{g'}$ représente le nombre d'appels en cours sur un résultat effectivement disponible g'. Les valeurs contenues dans les vecteurs $\overrightarrow{m}$ et $\overrightarrow{n}$ correspondent à celles du vecteur $\overrightarrow{y}$ de $\mathbb{P}'$. Elles sont séparées en deux en raison de la délégation qui apparaît ici et qui impose une étape supplémentaire. Ainsi, pour tout élément y_j du vecteur $\overrightarrow{y}$: $y_j = m_j + n_j$.

On définit maintenant la relation $\mathcal{S}$ telle que les couples suivant appartiennent à $\mathcal{S}$[9] :

$$\left\{\begin{array}{l} new\ done\left(\overline{done}\middle|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\right) \\ new\ done, newCell, exec, q, \overrightarrow{u}\Big(\\ \qquad \overline{done}\middle|\overline{newCell}\langle u_{f+1}\rangle\middle|\overline{exec}\langle u_{g+1}\rangle\middle|\mathbb{Q}'(a,b,c,d,e,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big) \end{array}\right. \tag{1}$$

$$\left\{\begin{array}{l} new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \left(\overline{r}\langle res_h\rangle.\overline{done}\middle|Object\langle res_h\rangle\right)\Big) \\ new\ done, newCell, exec, q, \overrightarrow{u}\Big(\\ \qquad \overline{newCell}\,\langle u_{f+1}\rangle\Big|\overline{exec}\,\langle u_{g+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\,\langle res,f_h\rangle\,\Big| \\ \qquad\qquad ActivCall.\overline{done}\Big| \\ \qquad\qquad \sum_i\Big\{m_i\,(\overrightarrow{p},r,act)\,.[[S_i]]\Big\}\Big)\Big) \end{array}\right. \tag{2}$$

$$\left\{\begin{array}{l} new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \left(\overline{r}\langle res_h\rangle.\overline{done}\middle|Object\langle res_h\rangle\right)\Big) \\ new\ done, newCell, exec, q, \overrightarrow{u}\Big(\\ \qquad \overline{newCell}\,\langle u_{f+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\,\langle res,f_h\rangle\,\Big| \\ \qquad\qquad ActivCall.\overline{done}\Big| \\ \qquad\qquad \sum_i\Big\{m_i\,(\overrightarrow{p},r,act)\,.[[S_i]]\Big\}\Big)\Big| \\ \qquad u_{g+1}\,((m_i,\overrightarrow{p},res),u')\,.new\ r'\,(\overline{m_i}\,\langle\overrightarrow{p},r',act_l\rangle\,.r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u'\rangle)\Big) \end{array}\right. \tag{3}$$

[9] Pour raccourcir les expressions on a abrégé $queue$ par q et $unqueue$ par u

$$
\left\{
\begin{array}{l}
new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}|Object\langle res_h\rangle\big)\Big) \\
new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\
\qquad \overline{newCell}\ \langle u_{f+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\ \langle res,f_h\rangle\ \Big| \\
\qquad\qquad ActivCall.\overline{done}\Big| \\
\qquad\qquad \sum_i\Big\{m_i\ (\overrightarrow{p},r,act)\ .[[S_i]]\Big\}\Big)\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\
\qquad\qquad \overline{m_i}\ \langle\overrightarrow{p},r,act_l\rangle\ .r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\
\qquad\qquad \sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)\Big)
\end{array}
\right. \tag{4}
$$

$$
\left\{
\begin{array}{l}
new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}|Object\langle res_h\rangle\big)\Big) \\
new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\
\qquad \overline{newCell}\ \langle u_{f+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\ \langle res,f_h\rangle\ \Big| \\
\qquad\qquad ActivCall.\overline{done}\Big| \\
\qquad\qquad \sum_i\Big\{m_i\ (\overrightarrow{p},r,act)\ .[[S_i]]\Big\}\Big)\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\
\qquad\qquad r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\
\qquad\qquad new\ res_h\ \big(Object\langle res_h\rangle|\overline{r'}\langle res_h\rangle\big)\Big)\Big)
\end{array}
\right. \tag{5}
$$

$$
\left\{
\begin{array}{l}
new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}|Object\langle res_h\rangle\big)\Big) \\
new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\
\qquad \overline{newCell}\ \langle u_{f+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\ \langle res,f_h\rangle\ \Big| \\
\qquad\qquad ActivCall.\overline{done}\Big| \\
\qquad\qquad \sum_i\Big\{m_i\ (\overrightarrow{p},r,act)\ .[[S_i]]\Big\}\Big)\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\
\qquad\qquad new\ res_h\Big(\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\
\qquad\qquad\qquad Object\langle res_h\rangle\Big)\Big)\Big)
\end{array}
\right. \tag{6}
$$

$$\left\{\begin{array}{l} new\ done\ \Big(\overline{done}\Big|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big) \\ new\ done, newCell, exec, q, \overrightarrow{u}\Big(\\ \qquad \overline{done}\Big|\overline{exec}\,\langle u_{g+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad queue(elt).\,\big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big)\end{array}\right. \tag{7}$$

$$\left\{\begin{array}{l} new\ done\ \Big(\overline{done}\Big|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big) \\ new\ done, newCell, exec, q, \overrightarrow{u}\Big(\\ \qquad \overline{done}\Big|\mathbb{Q}'(a,b,c,d,e,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad queue(elt).\,\big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big| \\ \qquad u_{g+1}\,((m_i,\overrightarrow{p},res),u').new\ r'\,(\overline{m_i}\,\langle\overrightarrow{p},r',act_l\rangle.r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u'\rangle)\Big)\end{array}\right. \tag{8}$$

$$\left\{\begin{array}{l} new\ done\ \Big(\overline{done}\Big|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big) \\ new\ done, newCell, exec, q, \overrightarrow{u}\Big(\\ \qquad \overline{done}\Big|\mathbb{Q}'(a,b,c,d,e,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad queue(elt).\,\big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad \overline{m_i}\,\langle\overrightarrow{p},r,act_l\rangle.r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad \textstyle\sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)\Big)\end{array}\right. \tag{9}$$

$$\left\{\begin{array}{l} new\ done\ \Big(\overline{done}\Big|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big) \\ new\ done, newCell, exec, q, \overrightarrow{u}\Big(\\ \qquad \overline{done}\Big|\mathbb{Q}'(a,b,c,d,e,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad queue(elt).\,\big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad new\ res_h\,\big(Object\langle res_h\rangle|\overline{r'}\langle res_h\rangle\big)\Big)\Big)\end{array}\right. \tag{10}$$

$$\left\{\begin{array}{l} new\ done\ \Big(\overline{done}\Big|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big) \\ new\ done, newCell, exec, q, \overrightarrow{u}\Big(\\ \qquad \overline{done}\Big|\mathbb{Q}'(a,b,c,d,e,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad queue(elt).\,\big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad new\ res_h\Big(\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad\qquad Object\langle res_h\rangle\Big)\Big)\Big)\end{array}\right. \tag{11}$$

$$\left\{\begin{array}{l} new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}|Object\langle res_h\rangle\big)\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \overline{exec}\,\langle u_{g+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e-1,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad queue(elt).\ \big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\,\langle res,f_h\rangle\ \Big| \\ \qquad\qquad ActivCall.\overline{done}\Big| \\ \qquad\qquad \sum_i\Big\{m_i\,(\overrightarrow{p},r,act)\,.[[S_i]]\Big\}\Big)\Big) \end{array}\right. \tag{12}$$

$$\left\{\begin{array}{l} new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}|Object\langle res_h\rangle\big)\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \mathbb{Q}'(a,b,c,d,e-1,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad queue(elt).\ \big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\,\langle res,f_h\rangle\ \Big| \\ \qquad\qquad ActivCall.\overline{done}\Big| \\ \qquad\qquad \sum_i\Big\{m_i\,(\overrightarrow{p},r,act)\,.[[S_i]]\Big\}\Big)\Big| \\ \qquad u_{g+1}\,((m_i,\overrightarrow{p},res)\,,u')\,.new\ r'\,(\overline{m_i}\,\langle\overrightarrow{p},r',act_l\rangle\,.r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u'\rangle)\Big) \end{array}\right. \tag{13}$$

$$\left\{\begin{array}{l} new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}|Object\langle res_h\rangle\big)\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \mathbb{Q}'(a,b,c,d,e-1,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad queue(elt).\ \big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\,\langle res,f_h\rangle\ \Big| \\ \qquad\qquad ActivCall.\overline{done}\Big| \\ \qquad\qquad \sum_i\Big\{m_i\,(\overrightarrow{p},r,act)\,.[[S_i]]\Big\}\Big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad \overline{m_i}\,\langle\overrightarrow{p},r,act_l\rangle\,.r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad \sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)\Big) \end{array}\right. \tag{14}$$

$$
\left\{\begin{array}{l}
new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}\big|Object\langle res_h\rangle\big)\Big) \\
new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\
\qquad \mathbb{Q}'(a,b,c,d,e-1,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\
\qquad queue(elt).\ \big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\ \langle res,f_h\rangle\ \Big| \\
\qquad\qquad ActivCall.\overline{done}\Big| \\
\qquad\qquad \sum_i\Big\{m_i\ (\overrightarrow{p},r,act)\ .[[S_i]]\Big\}\Big)\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\
\qquad\qquad r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\
\qquad\qquad new\ res_h\ \big(Object\langle res_h\rangle|\overline{r'}\langle res_h\rangle\big)\Big)\Big)
\end{array}\right. \tag{15}
$$

$$
\left\{\begin{array}{l}
new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}\big|Object\langle res_h\rangle\big)\Big) \\
new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\
\qquad \mathbb{Q}'(a,b,c,d,e-1,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\
\qquad queue(elt).\ \big(\overline{newCell}\langle u_{f+2}\rangle|\overline{u_{f+1}}\langle elt,u_{f+2}\rangle\big)\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\ \langle res,f_h\rangle\ \Big| \\
\qquad\qquad ActivCall.\overline{done}\Big| \\
\qquad\qquad \sum_i\Big\{m_i\ (\overrightarrow{p},r,act)\ .[[S_i]]\Big\}\Big)\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\
\qquad\qquad new\ res_h\Big(\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\
\qquad\qquad\qquad Object\langle res_h\rangle\Big)\Big)\Big)
\end{array}\right. \tag{16}
$$

$$
\left\{\begin{array}{l}
new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}\big|Object\langle res_h\rangle\big)\Big) \\
new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\
\qquad \overline{exec}\ \langle u_{g+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e-1,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\
\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\ \langle res,f_h\rangle\ \Big| \\
\qquad\qquad \overline{r}\langle f_h\rangle.\overline{done}\Big|\overline{newCell}\langle u_{f+2}\rangle\Big| \\
\qquad\qquad \overline{u_{f+1}}\ \langle(m_i,\overrightarrow{p},res),u_{f+2}\rangle\ \Big| \\
\qquad\qquad \sum_i\Big\{m_i\ (\overrightarrow{p},r,act)\ .[[S_i]]\Big\}\Big)\Big)
\end{array}\right. \tag{17}
$$

$$\left\{\begin{array}{l} new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}\big|Object\langle res_h\rangle\big)\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \mathbb{Q}'(a,b,c,d,e-1,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\,\langle res,f_h\rangle\ \Big| \\ \qquad\qquad \overline{r}\langle f_h\rangle.\overline{done}\Big|\overline{newCell}\langle u_{f+2}\rangle\Big| \\ \qquad\qquad \overline{u_{f+1}}\,\langle(m_i,\overrightarrow{p},res),u_{f+2}\rangle\ \Big| \\ \qquad\qquad \sum_i\Big\{m_i\,(\overrightarrow{p},r,act)\,.[[S_i]]\Big\}\Big)\Big| \\ \qquad u_{g+1}\,((m_i,\overrightarrow{p},res)\,,u')\,.new\ r'\,(\overline{m_i}\,\langle\overrightarrow{p},r',act_l\rangle\,.r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u'\rangle)\Big) \end{array}\right. \tag{18}$$

$$\left\{\begin{array}{l} new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}\big|Object\langle res_h\rangle\big)\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \mathbb{Q}'(a,b,c,d,e-1,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\,\langle res,f_h\rangle\ \Big| \\ \qquad\qquad \overline{r}\langle f_h\rangle.\overline{done}\Big|\overline{newCell}\langle u_{f+2}\rangle\Big| \\ \qquad\qquad \overline{u_{f+1}}\,\langle(m_i,\overrightarrow{p},res),u_{f+2}\rangle\ \Big| \\ \qquad\qquad \sum_i\Big\{m_i\,(\overrightarrow{p},r,act)\,.[[S_i]]\Big\}\Big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad \overline{m_i}\,\langle\overrightarrow{p},r,act_l\rangle\,.r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad \sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)\Big) \end{array}\right. \tag{19}$$

$$\left\{\begin{array}{l} new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}\big|Object\langle res_h\rangle\big)\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \mathbb{Q}'(a,b,c,d,e-1,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\,\langle res,f_h\rangle\ \Big| \\ \qquad\qquad \overline{r}\langle f_h\rangle.\overline{done}\Big|\overline{newCell}\langle u_{f+2}\rangle\Big| \\ \qquad\qquad \overline{u_{f+1}}\,\langle(m_i,\overrightarrow{p},res),u_{f+2}\rangle\ \Big| \\ \qquad\qquad \sum_i\Big\{m_i\,(\overrightarrow{p},r,act)\,.[[S_i]]\Big\}\Big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad new\ res_h\ \big(Object\langle res_h\rangle|\overline{r'}\langle res_h\rangle\big)\Big)\Big) \end{array}\right. \tag{20}$$

$$\left\{\begin{array}{l} new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \big(\overline{r}\langle res_h\rangle.\overline{done}\big|Object\langle res_h\rangle\big)\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \mathbb{Q}'(a,b,c,d,e-1,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res\langle\overrightarrow{m'}\rangle(\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\Big(Future\ \langle res,f_h\rangle\ \Big| \\ \qquad\qquad \overline{r}\langle f_h\rangle.\overline{done}\Big|\overline{newCell}\langle u_{f+2}\rangle\Big| \\ \qquad\qquad \overline{u_{f+1}}\ \langle(m_i,\overrightarrow{p},res),u_{f+2}\rangle\ \Big| \\ \qquad\qquad \sum_i\Big\{m_i\ (\overrightarrow{p},r,act)\ .[[S_i]]\Big\}\Big)\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad new\ res_h\Big(\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad Object\langle res_h\rangle\Big)\Big)\Big) \end{array}\right. \tag{21}$$

$$\left\{\begin{array}{l} new\ done\ \Big(\overline{done}\Big|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \overline{done}\Big|\overline{newCell}\ \langle u_{f+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e,f,g,0,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad u_{g+1}\ ((m_i,\overrightarrow{p},res)\ ,u')\ .new\ r'\ (\overline{m_i}\ \langle\overrightarrow{p},r',act_l\rangle\ .r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u'\rangle)\Big) \end{array}\right. \tag{22}$$

$$\left\{\begin{array}{l} new\ done\ \Big(\overline{done}\Big|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \overline{done}\Big|\overline{newCell}\ \langle u_{f+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad \overline{m_i}\ \langle\overrightarrow{p},r,act_l\rangle\ .r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad \sum_i\Big\{m_i(\overrightarrow{p},r,act).[[S_i]]\Big\}\Big)\Big) \end{array}\right. \tag{23}$$

$$\left\{\begin{array}{l} new\ done\ \Big(\overline{done}\Big|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big) \\ new\ done,newCell,exec,q,\overrightarrow{u}\Big(\\ \qquad \overline{done}\Big|\overline{newCell}\ \langle u_{f+1}\rangle\Big|\mathbb{Q}'(a,b,c,d,e,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\Big| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad r'(res_h).\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad new\ res_h\ \big(Object\langle res_h\rangle|\overline{r'}\langle res_h\rangle\big)\Big)\Big) \end{array}\right. \tag{24}$$

$$\left\{\begin{array}{l} new\ done\left(\overline{done}\middle|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\right) \\ new\ done, newCell, exec, q, \overrightarrow{u}\Big(\\ \qquad \overline{done}\middle|\overline{newCell}\,\langle u_{f+1}\rangle\middle|\mathbb{Q}'(a,b,c,d,e,f,g,1,\overrightarrow{x},\overrightarrow{l},\overrightarrow{m},\overrightarrow{n})\middle| \\ \qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},f_h,res,r'\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\langle\overrightarrow{a}\rangle\langle f_h\rangle\Big(Future\langle res,f_h\rangle\Big| \\ \qquad\qquad new\ res_h\Big(\overline{res}\langle res_h\rangle.\overline{exec}\langle u_{g+2}\rangle\Big| \\ \qquad\qquad\qquad Object\langle res_h\rangle\Big)\Big)\Big) \end{array}\right. \tag{25}$$

$\mathcal{S}$ est une bisimulation faible. Pour toute paire d'agents (A, B) appartenant à $\mathcal{S}$, si A se dérive, par α en A' ($A \xrightarrow{\alpha} A'$), alors il existe B' tel que B se dérive en B' par α et diverses τ-transitions ($B \stackrel{\alpha}{\Longrightarrow} B'$) et A' et B' appartiennent à $\mathcal{S}$. La figure 22 page suivante représente certaines transitions entre les processus $\mathbb{Q}$ des différentes équations. Les transitions qui ne modifient pas la formule du processus aux valeurs $a, b, c, d, e, f, g, \overrightarrow{x}, \overrightarrow{l}, \overrightarrow{m}, \overrightarrow{n}$ près ne sont pas représentées. Les ronds blancs correspondent aux processus en relation avec $\mathbb{P}_1$ et les ronds hachurés ceux en relations avec $\mathbb{P}_2$ avec :

$$\begin{array}{l} \mathbb{P}_1 = new\ done\left(\overline{done}\middle|\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\right) \\ \mathbb{P}_2 = new\ done\Big(\mathbb{P}'(a,b,c,d,e,f,\overrightarrow{x},\overrightarrow{y})\Big| \\ \qquad\qquad new\ \overrightarrow{m},\overrightarrow{m'},\overrightarrow{a},res_h\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p},r,act)\ \langle\overrightarrow{a}\rangle\ \left(\overline{r}\langle res_h\rangle.\overline{done}\middle|Object\langle res_h\rangle\right)\Big) \end{array}$$

La figure 23 page suivante représente les transitions entre les processus $\mathbb{P}_1$ et $\mathbb{P}_2$. De même que sur la figure précédente, les transitions ne modifiant pas la formule du processus aux valeurs de $a, b, c, d, e, f, g, \overrightarrow{x}, \overrightarrow{y}$ près ne sont pas représentées. On constate, grâce à ces deux figures que pour la τ-transition de $\mathbb{P}_1$ à $\mathbb{P}_2$, il existe un chemin de τ-transitions dans la figure 22 page suivante d'un rond blanc vers un rond hachuré. De même, pour la transition $\overline{r}$ de $\mathbb{P}_2$ à $\mathbb{P}_1$, il existe une suite de transitions $\tau^*.\overline{r}.\tau^*$ d'un rond hachuré vers un rond blanc dans la figure 22 page suivante. On peut faire la remarque symétrique, chaque fois qu'il existe une transition d'un rond blanc vers un rond hachuré sur la figure 22 page suivante, il s'agit d'une τ-transition et la τ-transition de $\mathbb{P}_1$ à $\mathbb{P}_2$ lui correspond. Toutes les transitions d'un rond hachuré vers un rond blanc dans la figure 22 page suivante correspondent à des transitions $\overline{r}$. On retrouve cette même transition sur la figure 23 page suivante entre $\mathbb{P}_2$ et $\mathbb{P}_1$.

Ces deux figures permettent de se convaincre que, en ce qui concerne les transitions ne modifiant pas les expressions des processus aux valeurs $a, b, c, d, e, f, g, \overrightarrow{x}, \overrightarrow{y}, \overrightarrow{l}, \overrightarrow{m}, \overrightarrow{n}$ près, $\mathcal{S}$ est bien une bisimulation faible. On peut s'assurer que c'est bien le cas pour toutes les autres transitions possibles en observant que les contraintes sur les valeurs de $a, b, c, d, e, f, g, \overrightarrow{x}, \overrightarrow{y}, \overrightarrow{l}, \overrightarrow{m}, \overrightarrow{n}$ sont toujours respectées. On rappellera également que tous les termes $[[S_i]]$ se terminent par le retour d'un résultat sur le canal r. Ainsi on peut

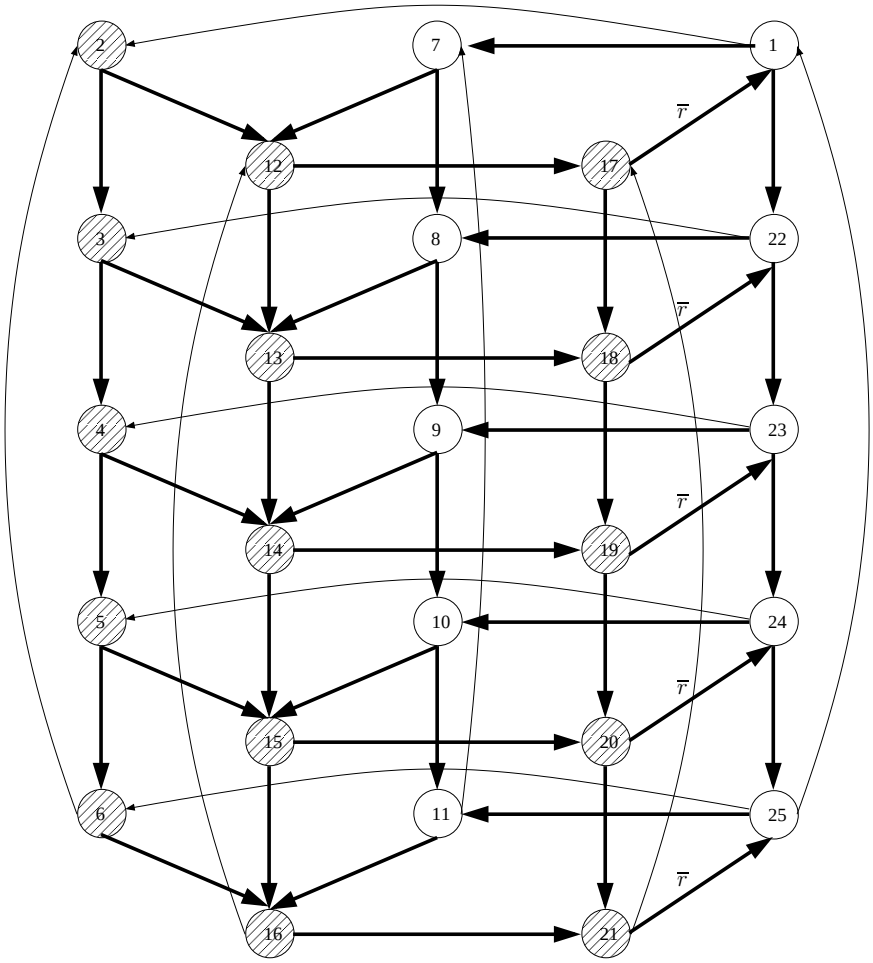


FIG. 22 – Transitions entre les processus $\mathbb{Q}$ des équations 1 à 25.

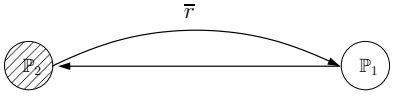


FIG. 23 – Transitions entre les processus $\mathbb{P}_1$ et $\mathbb{P}_2$.

considérer que :

$$new\ \overrightarrow{m}\ \langle\overrightarrow{m}\rangle\ (\overrightarrow{p}, r, act)[[S_i]] \approx new\ \overrightarrow{m}, \overrightarrow{m'}\ \langle\overrightarrow{m'}\rangle\ (\overrightarrow{p}, r, act)\ new\ r'\Big(r'(res_h).\overline{r}\langle res_h\rangle\Big| \{^{\overrightarrow{p}}/_{\overrightarrow{p}}\}\{^{r'}/_{r}\}\{^{act}/_{act}\}[[S_i]]\Big)$$

Finalement, $\mathbb{P}$ et $\mathbb{Q}$ sont en relation par $\mathcal{S}$. Ces deux processus sont donc en bisimulation faible. Il existe donc bien une bisimulation faible entre les processus P et Q. En d'autres termes, le processus modélisant un objet activable est en bisimulation faible avec le processus modélisant cet objet activé : $P \approx Q$. De ce fait :

1. $\alpha.P + M \approx \alpha.Q + M$
2. $new\ aP \approx new\ aQ$
3. $P|R \approx Q|R$
4. $R|P \approx R|Q$

Cela nous permet de conclure que tout ou partie des objets activables dans un programme peuvent être activés sans modifier le comportement de ce programme. En particulier pour 1, $\alpha = k_C$, le canal permettant la création de l'objet : un objet doit d'abord être créé avant de pouvoir être utilisé.

5.4 L'activabilité en pratique

Nous avons décrit une propriété d'activabilité que nous avons validée. Ainsi un objet respectant cette propriété est dit activable et peut être transformé afin que les appels sur cet objet soient effectués de façon asynchrone. Cette propriété permet, notamment, la mobilité d'objets entre sous-systèmes en évitant les problèmes d'étreinte fatale qui pourraient en découler. Pour autant l'activabilité reste, jusqu'à présent, une notion abstraite. Nous allons étudier maintenant les conséquences et la mise en œuvre pratique de cette propriété.

On peut considérer l'activabilité sous deux aspects distincts, celui d'un modèle de conception, assurant au développeur un certain nombre de garanties et celui d'un outil automatique de parallélisation d'applications. Dans le premier cas, c'est au développeur de se convaincre que, dans son application, un objet respecte le critère d'activabilité et de l'activer lui-même en sachant que le comportement de son programme ne sera pas altéré par cette modification. Nous allons étudier le second cas et les outils qui peuvent être mis en œuvre pour détecter automatiquement les objets activables dans un programme.

5.4.1 Détection des objets activables

La première difficulté posée par la mise en œuvre de l'activabilité étendue, mais également de la *TB*-activabilité, est la détection des objets activables dans un programme.

Nous nous intéressons ici aux outils permettant cette détection de façon automatique et en particulier aux méthodes d'analyse de programmes.

Les méthodes d'analyse de programmes ont pour objectif de détecter un certain nombre de caractéristiques ainsi que certains types de comportements des programmes de façon statique, c'est-à-dire sans exécuter le programme en question. On comprend naturellement que ce type d'analyse ne peut fournir que des informations partielles sur l'exécution du programme,[10] c'est pourquoi elles fournissent des approximations *sûres* des comportements possibles. L'analyse statique de programme a très majoritairement pour objectif l'optimisation de code. Il est par exemple possible de détecter les *expressions disponibles* à certains points du programme afin d'éviter de les recalculer, de déterminer les relations entre les blocs d'affectation et les blocs d'utilisation de variables afin de supprimer les blocs inutiles ou de déplacer des blocs les uns par rapport aux autres.

De nombreux autres types d'analyses sont envisageables. Chaque analyse doit fournir une réponse qui garantit, pour l'optimisation envisagée, que le programme est une cible adaptée. Ainsi, les optimisations effectuées ne seront pas toutes celles possibles mais seulement celles, parmi les possibles, pour lesquelles l'analyse aura déterminé avec certitude qu'elles n'altèrent pas la sémantique du programme. Dans le cadre de l'analyse de programmes à objets, un certain nombre d'analyses ont pour objectif d'identifier les types concrets des objets du programme afin d'optimiser les appels de méthodes sur ces derniers en remplaçant le mécanisme de sélection dynamique de méthode par un mécanisme statique. Pour plus d'informations sur l'analyse de programmes et une description didactique de celles-ci, des relations entre les différentes approches existantes ainsi que des algorithmes possibles de mise en œuvre, le lecteur est invité à consulter l'ouvrage de Nielson, Nielson et Hankin [101].

Une méthode d'analyse statique de programmes Java permettant notamment la détection d'objets *TB*-activables est proposée dans [65]. Cette méthode d'analyse est une extension au langage Java de l'analyse de forme proposée par Sagiv et al. [110] et repose sur la technique de l'interprétation abstraite [35]. Intuitivement, le principe général de cette analyse est de construire, pour un programme donné, un automate représentant l'ensemble de ses états. Les transitions dans cet automate correspondent à l'exécution des instructions du programme. A ce niveau, un certain nombre d'abstractions sont faites pour représenter tous les états possibles du programme avec un nombre fini d'états. Un graphe abstrait d'objets ainsi qu'une pile d'exécution abstraite sont associés à chacun des états de l'automate. Ils représentent la pile d'appel du programme à cet instant de son exécution ainsi que les objets présents en mémoire. Le graphe d'objets abstrait est composé d'un ensemble de nœuds représentant un ou plusieurs objets concrets. Ces nœuds sont associés à un prédicat permettant de savoir s'il sont partagés ou non, c'est-à-dire si plusieurs objets les référencent.

Enfin, la vérification de la propriété d'activabilité sur cette représentation abstraite de l'exécution du programme consiste à analyser le graphe d'objets abstrait de chacun des

[10] Sinon à quoi bon l'exécuter ?

états de l'automate pour y détecter des objets activables. Finalement, les objets considérés comme effectivement activables seront ceux activables dans tous les états de l'automate où ils apparaissent – c'est-à-dire activables durant toute leur existence.

Une telle analyse ne peut, en l'état, appréhender les programmes à processus légers en raison de l'explosion combinatoire qui apparaît lorsqu'on traite les interactions entre ces différents processus. Ceci est tout à fait en accord avec la *TB*-activabilité qui s'intéresse aux programmes séquentiels. Mais cette analyse ne peut pas être utilisée dans le cas de notre extension de l'activabilité aux programmes à processus légers.

Dans un autre registre, André Spiegel a proposé une analyse de programmes Java pour la distribution automatique d'applications dans le cadre de la plate-forme *Pangaea* [116]. L'objectif ici est d'identifier l'ensemble des objets d'une application, de les présenter ensuite au développeur sous la forme d'un graphe pour lui permettre de sélectionner un partitionnement de son application. Par la suite, la plate-forme se charge de générer le programme distribué correspondant. Cette analyse est insensible au flot d'exécution et peut donc traiter des applications composées de plusieurs processus léger. Le résultat produit est un graphe dont les sommets représentent un ou plusieurs objets et les arêtes entre ces sommets représentent les relations de création, d'utilisation ou simplement de référencement.

Nous avons utilisé cette plate-forme pour effectuer certaines expérimentations sur la détection des objets activables dans une application [56] mais sans intégrer la notion de potentiel d'utilisation que nous avons décrite plus haut. La plate-forme proposée ne permet pas la mobilité entre sous-systèmes. En effet, l'analyse proposée par Spiegel permet de s'intéresser aux applications à processus légers mais, en contrepartie, il devient impossible d'identifier des éléments dynamiques de l'exécution. Par exemple, le fait qu'un objet A cesse de référencer ou d'utiliser un objet X comme dans l'exemple de la figure 19 page 117, n'apparaîtra pas dans le graphe produit par *Pangaea* qui signalera uniquement que A utilise X au cours de l'exécution du programme.

D'autres types d'analyses sont proposés. Elles permettent de s'intéresser aux applications parallèles sans pour autant perdre toute information sur l'ordre d'exécution du programme. C'est par exemple le cas de l'approche proposée par Rinard et Whaley [144]. Cependant, même dans ce cas, un certain nombre d'imprécisions demeurent. Dans le cadre de l'optimisation de programmes, le manque de précision d'une analyse ou les différences de précision entre deux analyses n'ont généralement pas un impact important pour le développeur : partout où l'analyse en est capable, on améliore les performances. A l'inverse, l'activation d'un objet n'a pas forcément comme conséquence l'amélioration des performances. En pratique, le surcoût lié à un objet actif est très important. La décision d'activer ou non un objet relève donc d'un compromis qu'il est difficile de gérer automatiquement.[11] La mise en pratique de la propriété d'activabilité et de ce qu'elle permet –

[11] Nous ne pouvons bien sûr pas exclure la mise au point de futures techniques prenant parfaitement en compte ce genre de considérations. On en est cependant pas là aujourd'hui.

l'activation de certains objets de façon sûre – s'envisage donc aujourd'hui avec la collaboration du développeur. C'est la piste proposée dans la thèse de Romain Guider [65] et c'est celle que nous avions nous même choisie lors de nos premières expérimentations [56]. L'idée est qu'une analyse détecte un ensemble d'objets activables et propose au développeur de transformer son application pour les activer, ou encore de laisser le développeur écrire son application avec des objets actifs et lui confirmer si ses choix sont ou non judicieux par analyse du programme.

Dans le cas d'une optimisation de programme, le développeur n'a pas à se soucier des analyses qui peuvent être menées sur son programme. Quoiqu'il arrive, son application obtiendra (ou devrait obtenir) de meilleures performances que si rien n'avait été fait. Ici, au contraire, le développeur est directement informé des résultats de l'analyse. Il doit prendre certaines décisions en fonction de cela. Par exemple restructurer son application afin de permettre à l'analyse de *réaliser* qu'un objet est activable alors qu'elle ne l'avait pas détecté dans un premier temps. Cela implique, de la part du développeur, une connaissance en profondeur de l'analyse utilisée. Ceci ne constitue pas forcément une simplification de son travail, d'autant plus si on imagine que les analyses peuvent changer dans le temps et que cela demandera au développeur de s'adapter à nouveau. D'un autre côté, le fait que l'analyse puisse indiquer au développeur qu'elle ne peut pas garantir qu'un objet dont il a demandé l'activation soit effectivement activable, forcera le développeur à vérifier son application afin de se convaincre que son objet est bien activable même si l'analyse ne peut le détecter. On peut penser qu'il s'agit là d'une garantie supplémentaire en ce qui concerne la qualité du logiciel développé. Pourtant, le nombre important de fausses alertes qui risque d'apparaître nous laisse penser que cela incitera, au contraire, le développeur à négliger de plus en plus celles-ci.

Dans un tel mode de fonctionnement, on est finalement amené à s'interroger sur la pertinence de l'utilisation de l'analyse statique de programme dans le cadre du développement d'applications asynchrones.

5.4.2 *Séparabilité* : un peu de dynamisme

De façon générale, la propriété d'*activabilité* étendue telle que nous l'avons définie manque de dynamisme. En effet un objet, pour être activable, doit être activable tout au long de son existence dans le programme. Or cette caractéristique peut évoluer, un objet peut être activable pendant une durée donnée de l'exécution du programme puis ne plus l'être ou, inversement, ne pas être activable lors de sa création puis le devenir.

En particulier, un objet activable ne peut être utilisé que par un seul processus léger à la fois. Dans ce contexte, nous avons proposé la propriété de séparabilité [26]. Cette propriété défini qu'un objet est séparable s'il satisfait la propriété d'activabilité de la définition 5.8 page 118 sauf en ce qui concerne les processus légers. Autrement dit, s'il est *TB*-activable :

Définition 5.12. *Séparabilité*
Un objet o est séparable si et seulement si, à chaque instant de l'exécution du programme, pour tout objet o' dans l'accessibilité de o, pour tout objet o'' :

- *soit o'' appartient à l'accessibilité de o,*

- *soit o appartient à tous les chemins depuis o'' jusqu'à o'.*

Cette propriété nous permet d'introduire du dynamisme dans le caractère activable d'un objet. Ainsi, un objet peut être séparable, puis activable lorsqu'il peut être établi qu'un seul processus léger l'utilise et, à nouveau, séparable s'il apparaît que cet objet est potentiellement utilisé par plusieurs processus légers. La figure 24 page suivante illustre les transitions possibles entre ces différents états. Lorsqu'un objet est *séparé*, les appels sont effectués de façon synchrone depuis l'extérieur de son sous-système. Il peut devenir activable lorsqu'il peut être établi qu'il n'est utilisé que par un seul processus léger. Dans ce cas, il est *activé* et les appels deviennent asynchrones et sont effectués dans l'ordre *FIFO*, comme nous l'avons déjà décrit. Par la suite, s'il apparaît que l'objet actif est susceptible d'être partagé par plusieurs processus légers alors celui-ci n'est plus activable et redevient donc séparable. Dans ce cas, tous les nouveaux appelants sur l'objet sont bloqués le temps que tous les appels dans la file de l'objet actif aient été exécutés. L'objet passe ensuite dans l'état séparé et les appels bloqués ainsi que tous les appels suivant sont exécutés de façon synchrone.

Lors de son mémoire de DEA, Christophe Popov a développé le prototype d'une plateforme permettant la gestion de la notion de séparabilité [105]. Ce prototype a été développé au dessus de ProActive. Il ne permet pas le retour de l'état séparable ou activable vers l'état non-séparable mais les autres transitions présentées figure 24 page suivante sont gérées. Le fait qu'un objet séparable ne soit utilisé que par un seul processus léger n'est pas simple à établir. C'est pourquoi ce prototype repose sur les informations fournies par le développeur.

La propriété de séparabilité introduit un peu de dynamisme par rapport à la seule activabilité. Pourtant, même si un objet peut être actif ou non de manière alternée pendant l'exécution d'un programme, le fait de pouvoir être activable reste un élément statique dans le sens où cela dépend directement de la structure intrinsèque de l'application. Par exemple, pour qu'un objet puisse être activé, il est nécessaire que celui-ci soit une articulation ou une feuille du graphe. Cela dépend directement de la structure de l'application et limite donc fortement la quantité d'asynchronisme qui peut être introduite. Dans la section suivante nous présentons les bouquets d'activations qui permettent de restructurer le programme afin de faire apparaître de l'asynchronisme là où l'activabilité ne peut en apporter.

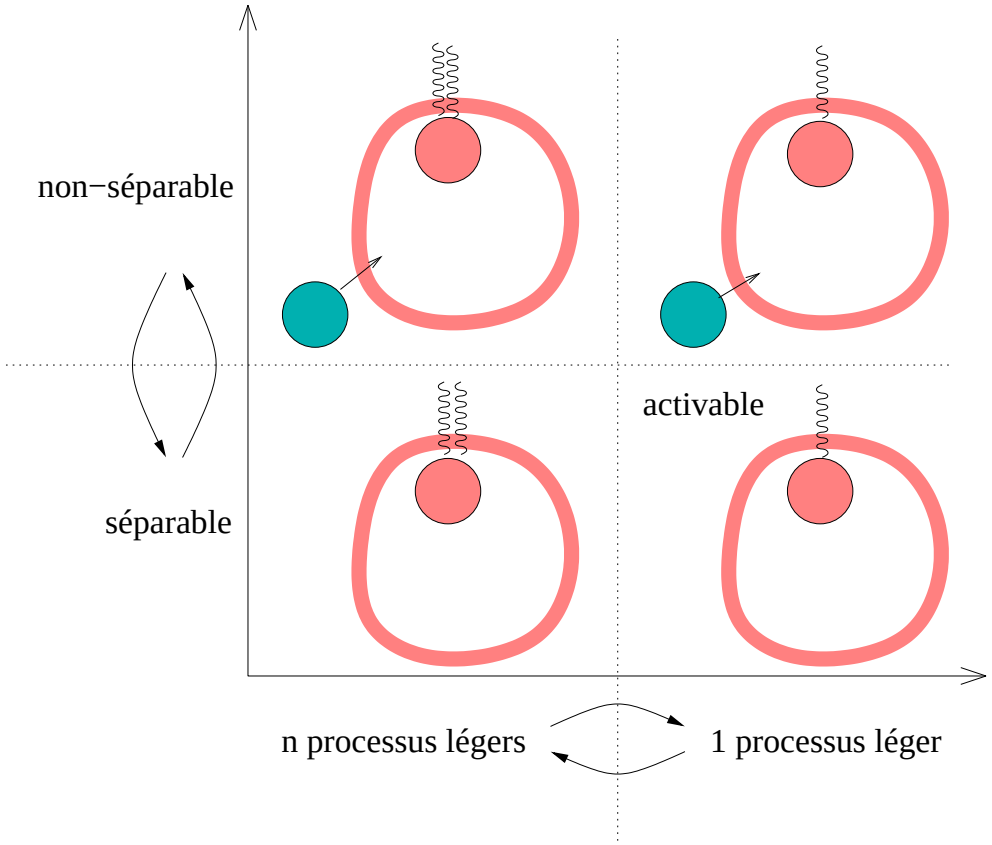


FIG. 24 – Transitions entre les états non séparable, séparable et activable.

5.5 Les bouquets d'activations

Les bouquets d'activations permettent de lever la limitation de l'activabilité liée à la structure du programme que nous venons de décrire. Ce concept est présenté ici dans un état préliminaire. L'étude approfondie de ce dernier fait l'objet des perspectives de ces travaux.

Plutôt que de considérer uniquement des objets et de détecter ceux qui sont activables, le principe des bouquets d'activations s'intéresse à des ensembles d'objets – les activations – possédant des propriétés spécifiques vis-à-vis du graphe d'accessibilité de la définition 5.7 page 117 et aux relations entre ces ensembles au travers des *bouquets d'activations* :

Définition 5.13. *Activation*
Soit A un ensemble d'objets, A est une activation si et seulement si, pour tout objet o appartenant à A, l'ensemble des objets de l'accessibilité de o est inclue dans A.

Définition 5.14. *Bouquet d'activations*
Soit $\mathcal{A}$ un ensemble d'activations, $\mathcal{A}$ est un bouquet d'activations si et seulement si, pour toute paire d'activations A et B appartenant à $\mathcal{A}$:

- *soit $A = B$;*
- *soit $A \cap B = \emptyset$;*
- *soit $A \cap B = A$, on dira alors que A est une sous-activation de B ;*
- *soit $A \cap B = B$, on dira alors que B est une sous-activation de A.*

Définition 5.15. *Activation de*
L'activation d'un objet o, notée $\mathbb{A}(o)$ est la plus petite activation A telle que $o \in \mathbb{A}(o)$.

De par les relations entre activations et sous-activations, on voit naturellement apparaître une structure arborescente des activations dans un bouquet. La figure 25 présente un graphe d'accessibilité et différents découpages en bouquets d'activations possibles. On constate par exemple que sur les quatre figures, les objets D et F sont toujours dans la même activation. En effet F est dans l'accessibilité de D et D est dans l'accessibilité de F. On doit donc avoir, d'après la définition 5.13 : $F \in \mathbb{A}(D)$ et $D \in \mathbb{A}(F)$. Autrement dit : $\mathbb{A}(F) \subseteq \mathbb{A}(D)$ et $\mathbb{A}(D) \subseteq \mathbb{A}(F)$, c'est-à-dire $\mathbb{A}(F) = \mathbb{A}(D)$. En outre, les objets B et C partagent l'objet D, donc $D \in \mathbb{A}(B) \cap \mathbb{A}(C)$. Ainsi, d'après cette même définition, soit $\mathbb{A}(B) = \mathbb{A}(C)$, comme c'est le cas sur les figures 25(c) et 25(d), soit $\mathbb{A}(B)$ est une sous-activation de $\mathbb{A}(C)$ ($\mathbb{A}(B) \subset \mathbb{A}(C)$) comme sur la figure 25(a), soit l'inverse comme sur la figure 25(b). Quand à l'objet E il doit forcément être dans la même activation que D (figure 25(d) page suivante) ou dans une sous-activation (figures 25(a), 25(b) et 25(c)).

Nous avons vu en 5.1.1 page 111 que le découpage d'un graphe d'accessibilité en sous-systèmes avec l'activabilité était unique. Les bouquets d'activations nous offrent plus de souplesse puisque différents bouquets peuvent être construits pour un même graphe d'accessibilité. L'intérêt est alors, comme pour l'activabilité, de rendre les appels de méthodes entre activations asynchrones, sans modifier la sémantique de l'application. Le

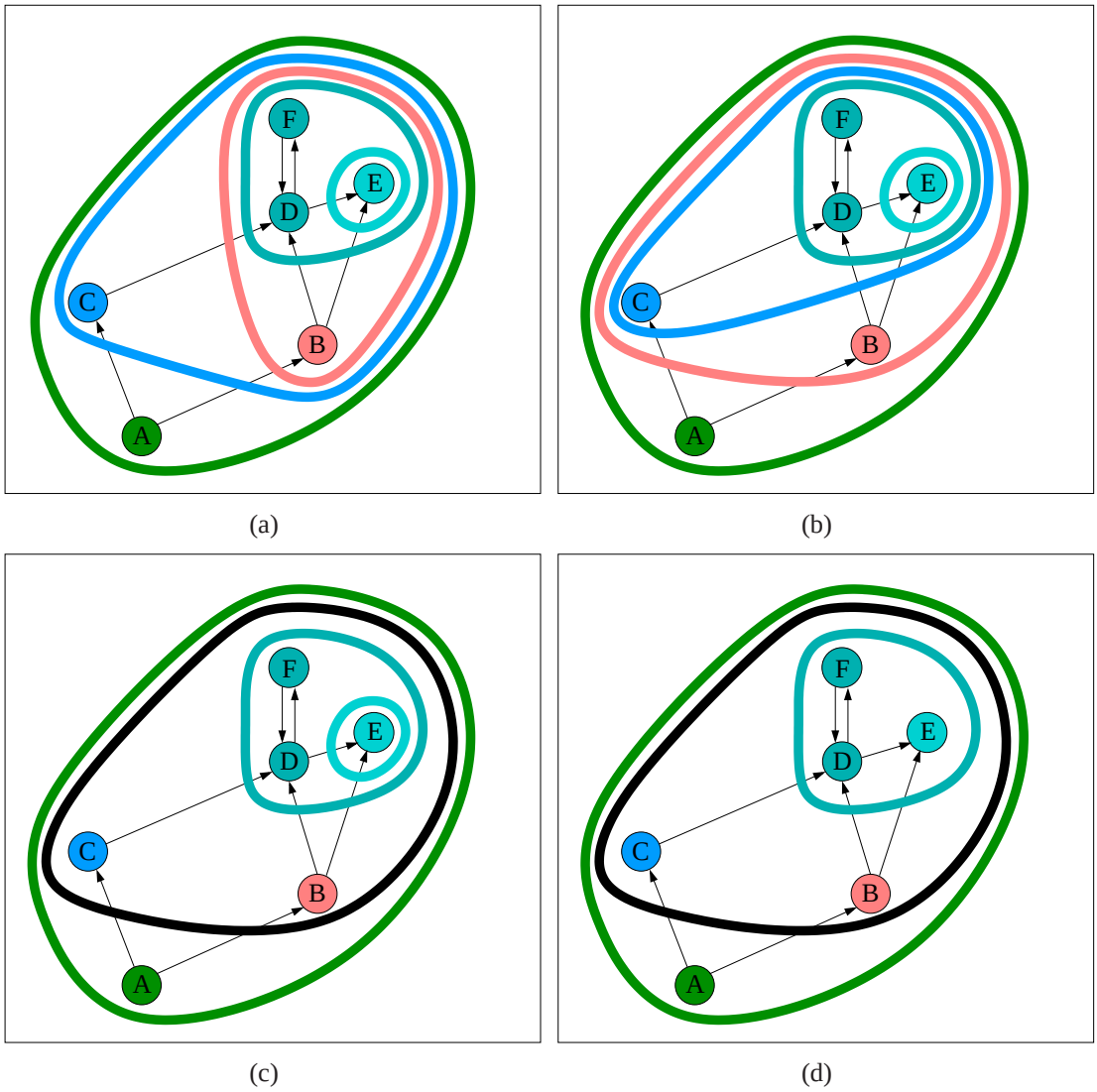


FIG. 25 – Différents bouquets d'activations possibles pour un même graphe d'objets.

fonctionnement dans ce cas est le suivant : les appels de l'extérieur d'une activation sont asynchrones, un futur est immédiatement retourné à l'appelant et l'appel est placé dans une file. Chaque activation ne possède qu'une seule file. Un processus, associé à cette activation est chargé d'exécuter les appels stockés dans la file dans l'ordre *FIFO*. Par exemple sur la figure 25(d) page ci-contre, les appels de B vers E seront asynchrones ainsi que les appels de B vers D. Par contre un seul processus sera chargé d'exécuter tous ces appels et ils le seront dans l'ordre *FIFO*, c'est-à-dire l'ordre dans lequel B aura effectué ces appels. Par ailleurs, un appel est susceptible de *traverser* plusieurs activations. Sur la figure 25(b), les appels de B vers E *traversent* trois activations. Dans ce cas, l'appel sera propagé par chacune des activations. C'est-à-dire qu'il sera d'abord stocké dans la file de la première activation. Lorsque le processus de cette activation prendra en charge cet appel, il le transmettra à l'activation suivante en le stockant dans sa file et ainsi de suite jusqu'à ce que l'activation de l'objet cible soit atteinte et l'appel effectivement exécuté.

Nous validerons ce mécanisme section 5.5.2 page 158. Cette validation repose directement sur celle de l'activabilité que nous avons présentée section 5.3 page 118.

5.5.1 Dynamisme des bouquets d'activations

Nous avons présenté les bouquets d'activations et l'asynchronisme qu'ils permettent d'obtenir dans une application. Dans le cas de l'activabilité, seuls des objets peuvent être activés et pour cela, il est nécessaire, entre autres, que ces objets constituent des points d'articulation ou des feuilles du graphe d'accessibilité du programme tout au long de l'exécution de ce programme. Les bouquets d'activations sont plus souples dans la mesure où il existe plusieurs bouquets d'activations possibles pour un même programme. Cependant, ce principe n'est pas encore dynamique. Nous allons voir maintenant comment on peut envisager le déplacement, au fur et à mesure de l'exécution d'un programme, d'objets entre les différentes activations de ce programme.

L'objectif est ici de définir un certain nombre de règles permettant de faire évoluer dynamiquement les activations d'un programme tout en s'assurant, d'une part que le programme est toujours structuré en bouquet et, d'autre part que la sémantique de l'application est bien respectée. L'utilisation en pratique des activations ne nécessite alors plus l'analyse préalable du programme. On peut envisager une utilisation systématique : chaque objet créé l'est dans une activation qui lui est propre. On peut également envisager de fournir une primitive au développeur pour créer une activation et y placer un objet : l'accessibilité de cet objet – calculable dynamiquement par un parcours en profondeur du graphe d'objets – est placée dans cette nouvelle activation. Dans tous les cas, le programme doit ensuite évoluer pour conserver une structure en bouquet. Parmi les règles possibles permettant une telle évolution, nous proposons les suivantes que nous validerons à la prochaine section :

1. Passage d'une référence sur un objet d'une activation A à un objet d'une sous-activation de A (invocation de méthode, affectation de champs) $\mapsto$ destruction de toutes les sous-activations de A contenant l'objet cible.

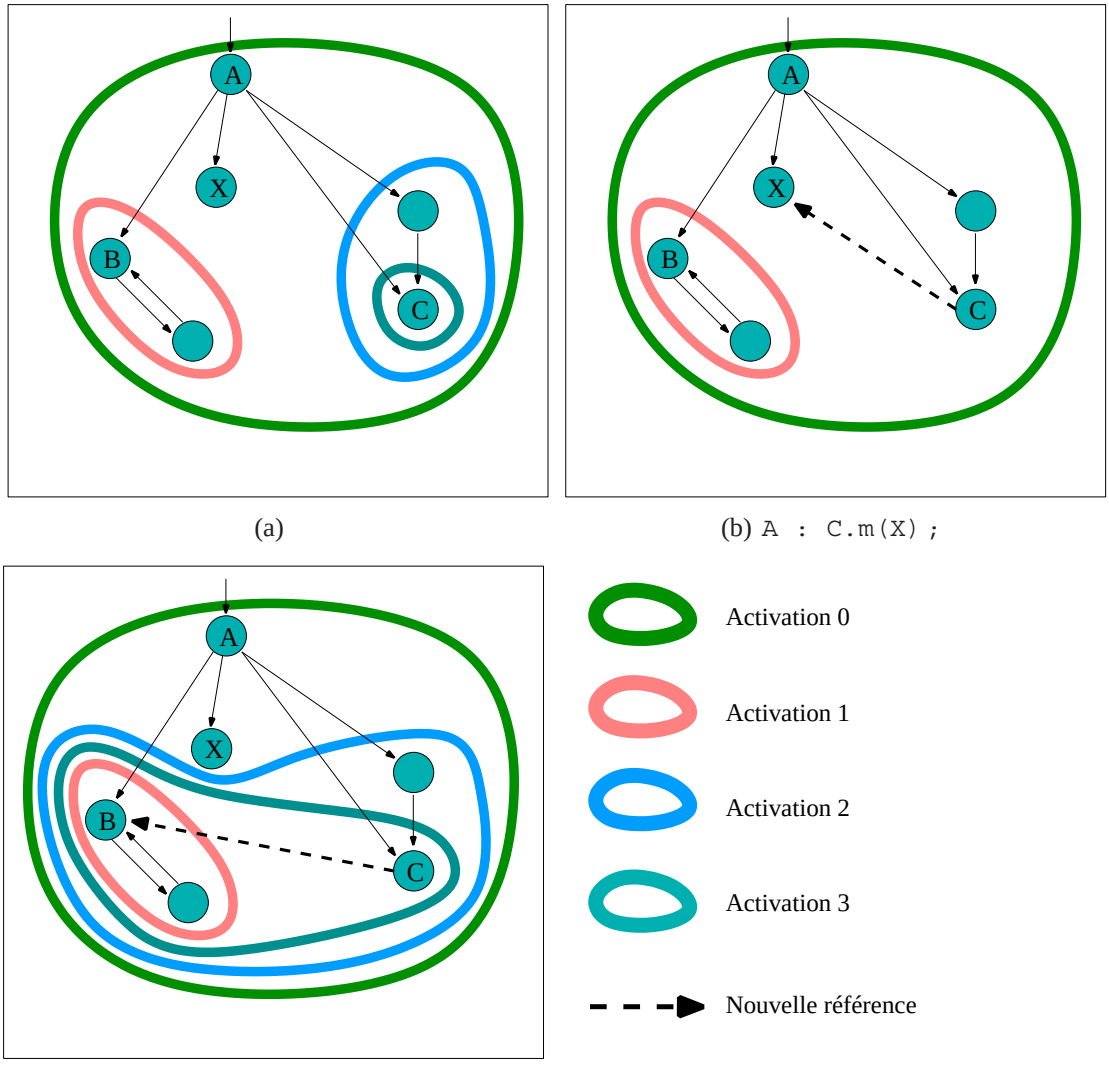


FIG. 26 – Transformation des activations.

Fonctionnement La destruction d'une activation exige qu'on attende d'abord que tous les appels de la file correspondante aient été exécutés. Suite à la destruction d'une sous-activation, les objets qu'elle contient appartiendront désormais à l'activation englobante. Si l'opération à l'origine de cette modification provient de l'activation A, alors celle-ci est effectuée de façon synchrone (puisque cela devient une opération intra-activation).

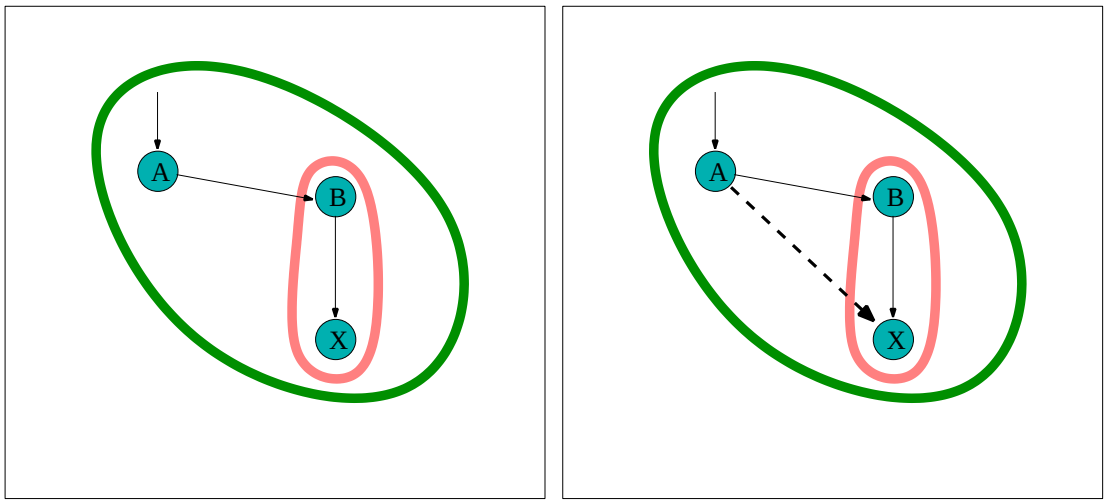
Exemple Sur la figure 26(a) page précédente, lorsque l'objet A passe l'objet X en argument à la méthode `m()` de l'objet C, cette règle s'applique et on obtient les activations représentées figure 26(b). On constate que les sous-activations 2 et 3 ont été détruites. L'appel a été fait de façon synchrone puisqu'il devient, après ces destructions, interne à l'activation 0.

Motivation Le passage de référence vers une sous-activation peut aboutir à un cycle entre l'objet cible et l'objet transmit. L'objet transmit, l'objet cible ainsi que tous les objets référencés par l'objet cible et l'objet transmit doivent donc être placés dans la même activation. Une autre possibilité serait de *plonger* l'accessibilité de l'objet transmit dans l'activation de l'objet cible. Cependant, dans le cas où la méthode d'un objet transmit serait présente sur la pile – c'est-à-dire en cours d'exécution – au moment de l'opération, il faudrait *reprendre* l'exécution en cours de cette méthode dans l'activation destination. Nous ne sommes pas en mesure actuellement de garantir la validité d'une telle transformation.

2. Passage d'une référence sur un objet d'une sous-activation A à un objet d'une sous-activation B telles que $A \cap B = \emptyset \mapsto$ la plus grande sous-activation C contenant la sous-activation A et telle que $C \cap B = \emptyset$ devient une sous-activation de la sous-activation B.

Exemple Cette règle s'applique entre les figures 26(a) et 26(c) page ci-contre où l'objet B, appartenant à l'activation 1 a été passé en argument à la méthode `m()` de l'objet C par l'objet A de l'activation 0. Dans ce cas, la plus grande sous-activation – de l'activation 0 – contenant l'activation 1 et ayant une intersection vide avec l'activation 3 est l'activation 1 elle-même. Cette dernière devient alors une sous-activation de l'activation 3.

Motivation Cette règle impose que ce soit la plus grande sous-activation C contenant la sous-activation A telle que $C \cap B = \emptyset$ qui devienne une sous-activation de B. On pourrait penser qu'il suffirait que seule la sous-activation A devienne une sous-activation de B mais cela casserait la structure en bouquet. En effet, si $C \neq A$ alors les activations B et C partageraient l'activation A sans être inclues l'une dans l'autre.



(a) (b) `B : return X;`

FIG. 27 – Retour de résultat

3. Retour de résultat $\mapsto$ rien ne se passe

   Exemple Entre les figures 27(a) et 27(b), une méthode de l'objet B retourne une référence vers l'objet X qui sera dorénavant référencé par A depuis l'extérieur de l'activation.

   Motivation Dans un programme structuré en bouquet, le résultat d'une méthode appartient toujours à l'activation ou à une sous-activation de l'appelé. De plus, un objet de l'activation courante – l'appelant dans ce cas – peut toujours référencer un objet d'une sous-activation sans casser le modèle en bouquet. Nous ne nous occupons pas ici de faire *remonter* le résultat d'une méthode lorsque c'est possible. C'est la règle suivante qui s'en chargera.

4. Ramasse-miettes montant : une composante fortement connexe n'est plus référencée depuis l'intérieur de son activation courante $\mapsto$ cette composante est *remontée* dans l'activation englobante.

   Exemple L'objet B (qui constitue une composante fortement connexe) de la figure 28(a) page ci-contre n'est pas référencé depuis l'intérieur de son activation. Il peut donc être *remonté* dans son activation englobante comme sur la figure 28(b).

   Motivation Nous proposons le terme *montant* pour cette opération de ramasse-miettes puisqu'elle a pour effet de faire *remonter* un élément d'une sous-activation vers son activation englobante. Cette opération permet de diminuer le nombre d'activations traversées par une référence. Son application systématique aura bien sûr

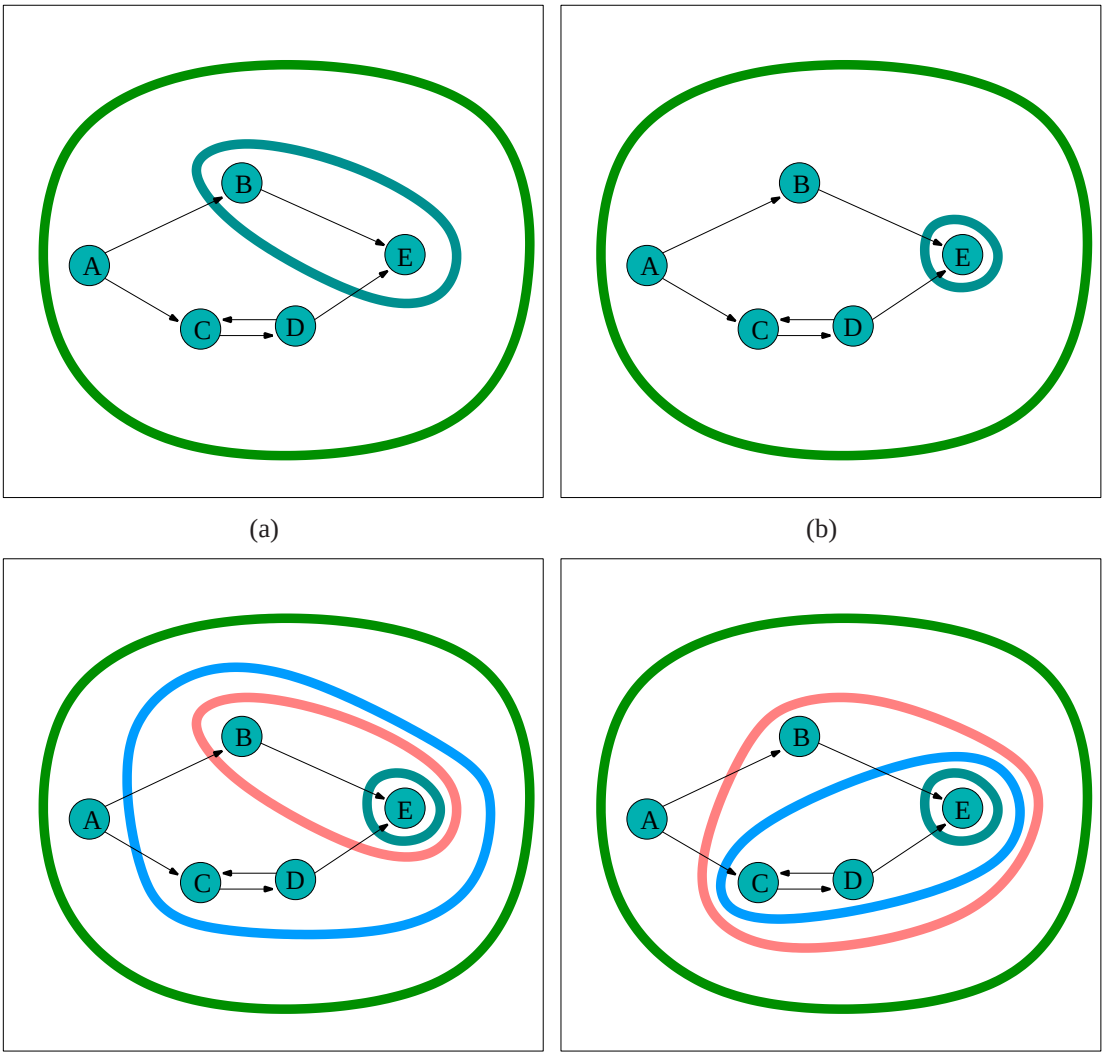


FIG. 28 – Opérations de ramasse miettes.

pour effet de rassembler tous les nœud dans la même activation. Sur la figure 28(b) page précédente, par exemple, on pourrait faire remonter E. Dans cette règle, on considère une composante fortement connexe et pas seulement un sommet car tous les sommets d'une même composante fortement connexe doivent être situés dans la même activation si on souhaite respecter la structure en bouquets.

5. Ramasse-miettes descendant : soit A une activation, C une composante fortement connexe de A ($\forall c \in C, \mathbb{A}(c) = A$) telle que, aucun sommet de C ne référence de sommet v tel que $\mathbb{A}(v) = A$ et $v \notin C \mapsto$ création d'une nouvelle sous-activation dans laquelle sont placés les sommets de C ainsi que toutes les activations référencées par ces sommets.

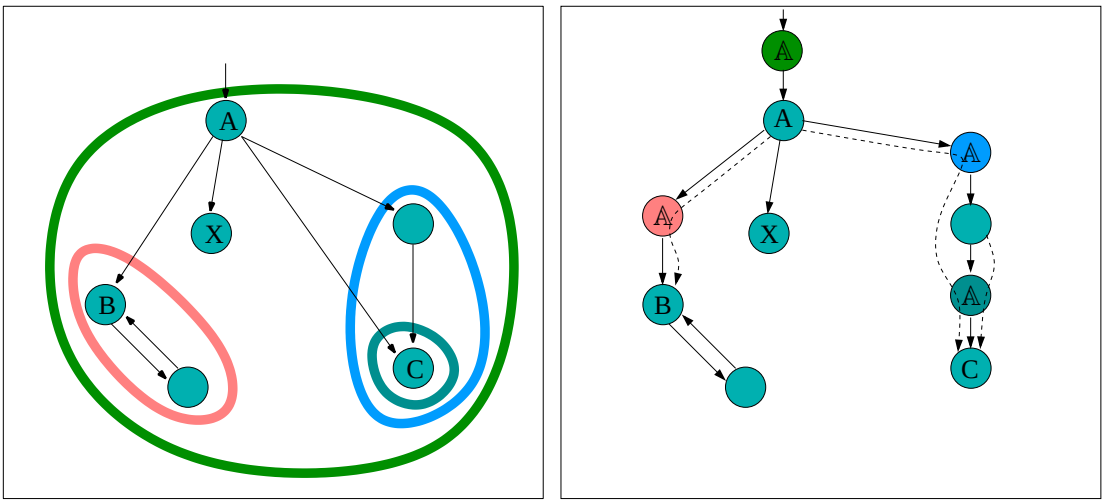
   Exemple Sur la figure 28(b) page précédente, l'objet B (qui constitue une composante fortement connexe) ne référence aucun sommet dont l'activation soit identique à son activation. Cet objet ainsi que l'activation de E qu'il référence peuvent donc être placés dans une nouvelle sous-activation. Par la suite, on constate que la composante fortement connexe formée des sommets C et D peut également être placée dans une nouvelle sous-activation. On obtient le bouquet de la figure 28(c).

   Motivation Cette règle permet d'introduire de nouvelles activations dans le programme. Par exemple la règle 1 page 153 détruit des activations qui peuvent être recréées grâce à cette règle. Cette règle ne s'applique qu'à une seule composante fortement connexe d'une activation donnée à la fois. On peut noter que le découpage d'une activation en sous-activations par un ramasse-miette descendant est sujet à différents choix. En effet, si deux composantes fortement connexes d'une même activation sont éligibles et qu'elles référencent des sous-activations communes, le résultat produit pourra varier selon l'ordre dans lequel elles seront traitées. Dans l'exemple précédent, si la composante fortement connexe constituée des sommets C et D est traitée avant B alors on obtient le bouquet de la figure 28(d) page précédente.

Ces règles sont compatibles avec la définition de bouquet d'activations que nous avons donnée puisqu'à chaque instant du programme, ce dernier est bien structuré en bouquet. Nous allons étudier dans la section suivante la validité des bouquets d'activations et de ces règles. D'autres règles respectant la structure en bouquet pourraient probablement être proposées. On peut noter que l'asynchronisme d'une application structurée en bouquet dépendra directement de la façon dont seront appliquées les règles 4 page 156 et 5 de ramasse-miettes.

5.5.2 Validité des bouquets d'activations

Nous allons maintenant montrer pourquoi un bouquet d'activations respecte la sémantique séquentielle d'un programme. Pour cela, nous nous appuyons directement sur



(a) Avant (b) Après

FIG. 29 – Première transformation

la validité de l'activabilité étendue présentée en 5.3 page 118 et nous proposons une transformation d'un programme séquentiel structuré en bouquet d'activations en un autre programme séquentiel équivalent dans lequel nous introduisons des objets spéciaux représentant chacun une activation du programme d'origine. Nous montrons ensuite que chacun de ces nouveaux objets est activable et que la mobilité des objets entre activations induite par les règles précédentes est conforme à la mobilité entre sous-systèmes que permet l'activabilité étendue.

Nous donnons ici l'intuition de la première transformation, celle-ci demande à être formalisée. La figure 29(b) représente l'application de cette transformation au programme en bouquet de la figure 29(a) : pour chaque activation, un objet spécial est introduit – notés $\mathbb{A}$ sur la figure 29(b). Toutes les références de l'extérieur de l'activation vers l'intérieur de celle-ci sont redirigées pour référencer le nouvel objet. Ce dernier référence lui-même tous les objets qui étaient, auparavant, référencés de l'extérieur de l'activation. De plus, cet objet possède un comportement particulier : pour chaque appel effectué sur ce dernier, il retransmet l'appel en question vers l'objet qui en était la cible dans le programme d'origine – représenté par les flèches en pointillé sur la figure. Ainsi, le principe de cette transformation est de centraliser tous les appels qui ont pour cible une même activation vers un seul et même objet qui les délègue ensuite. De plus ces objets spéciaux proposent des opérations afin de modifier l'ensemble des objets qu'ils référencent. Le programme transformé exploite ces opérations pour appliquer les règles définies dans la section précédente lorsqu'un appel de méthode se fait via un objet spécial. Cette transformation, même si elle demande encore à être formalisée, conserve la sémantique de l'application puisqu'il s'agit simplement d'un mécanisme de délégation.

Si on considère cette transformation possible on constate, d'après la définition 5.14 page 151 des bouquets d'activations, que, dans le nouveau graphe d'accessibilité obtenu,

tous les chemins de l'extérieur vers l'intérieur des accessibilités des nouveaux objets introduits passent par chacun d'entre eux. Ainsi, d'après la définition 5.8 page 118 et grâce à la mobilité entre sous-systèmes qu'elle autorise, ces objets sont activables.[12] La transformation du nouveau programme obtenu en un programme dans lequel tous les objets spéciaux sont actifs conserve donc bien la sémantique de l'application.

Il existe donc une suite de deux transformations qui, pour un programme décomposé en un bouquet d'activations donné, produit un programme à objets actifs respectant la sémantique du programme séquentiel d'origine et dont chacun des objets actifs *modélise* une activation. Ceci nous permet de dire que les bouquets d'activations permettent d'obtenir un programme concurrent dont la sémantique est équivalente à celle du programme séquentiel d'origine.

5.5.3 Perspectives

Dans cette section, nous avons décrit les bouquets d'activations. La structure arborescente des activations dans un bouquet permet de gérer de façon automatique les contraintes de synchronisation et d'ordre dans l'accès à des objets partagés. Cela nous permet d'introduire de la concurrence dans un programme et nous avons vu comment cette concurrence respecte la sémantique séquentielle du programme. On peut comparer cette approche à celle de la plate-forme Athapascan-1 [40] dans laquelle une application est composée d'un ensemble de tâches partageant des données. L'accès par les différentes tâches à ces données est réalisé en fonction des dépendances entre ces tâches et en respectant l'ordre séquentiel. Ces dépendances doivent être décrites par le développeur qui indique les données manipulées par une tâche. Ainsi, il est possible de spécifier les modes d'accès à une donnée, par exemple en lecture seulement, ou en écriture par accumulation, dans ce cas une fonction associative et commutative est appliquée à la donnée par différentes tâches dans un ordre quelconque. Dans le cas des bouquets d'activations, une dépendance est présente dès qu'un objet en référence un autre. En contrepartie de ce peu de finesse, ces dépendances sont déterminées sans l'intervention du programmeur. En outre, une tâche Athapascan-1 ne peut pas accéder aux résultats produits par une tâche fille qu'elle *lance*. Ici, au contraire, l'asynchronisme se faisant au niveau des appels de méthode, la récupération du résultat d'un appel de méthode, même asynchrone, est évidemment possible.

Bien que la validation des bouquets d'activations puisse être davantage formalisée, cette notion de bouquet constitue une base intéressante pour la mise au point d'un mécanisme transparent d'appels de méthode asynchrones puisque le programme concurrent respecte la sémantique séquentielle. Cependant, nous avons vu section 4.4.1 page 101 les limites de l'asynchronisme transparent, en particulier en ce qui concerne la gestion

[12] Pour faciliter les choses, on suppose ici que le programme n'a pas recours aux processus légers. Ceci nous évite de gérer le cas d'un objet utilisé par plusieurs processus légers qui changerait d'activation. On risquerait alors, après transformation, d'avoir un objet activable utilisé par plusieurs processus légers, ce qui est contraire à la définition 5.8 page 118.

des exceptions. C'est pourquoi nous privilégions la mise au point d'une plate-forme dont l'asynchronisme soit explicite. La syntaxe précise du langage associé doit être définie mais le principe est celui d'une syntaxe prenant en compte les spécificités de l'asynchronisme – pas d'exception levée au moment d'un appel de méthode ; récupération d'un futur – mais d'une sémantique correspondant à une exécution séquentielle. Ainsi, certains appels seront effectivement asynchrones tandis que d'autres resteront synchrones, soit pour des raisons de sémantique – appelant et appelé situés dans la même activation – soit pour des raisons pratiques d'adaptation à la plate-forme sous-jacente – nombre limité de processeurs. Le degré de concurrence est laissé à la discrétion de la plate-forme d'exécution sans impact sur la sémantique. En ce qui concerne la synchronisation entre appelant et appelé et l'utilisation du résultat d'un appel asynchrone, on peut noter que les bouquets d'activations permettent d'envisager l'utilisation anticipée de résultat plutôt que l'attente par nécessité. En effet, le résultat d'une méthode appartient toujours à l'activation de l'appelé, tout comme le résultat d'une méthode appartient au *vat* de l'appelé dans la plate-forme E (Cf. section 4.2.3 page 98). Il est donc possible de soumettre les appels au futur résultat vers cette activation sans attendre le résultat lui-même.

La transformation de programme que nous avons décrite en 5.5.2 page 158 établit une correspondance entre activation et objet activable. Selon la définition 5.8, un seul processus léger ne doit pouvoir accéder à un objet activable depuis l'extérieur de son sous-système. Il en est donc de même pour une activation : un seul processus léger ne doit pouvoir accéder aux objets d'une activation depuis l'extérieur de cette dernière. Cependant, une activation peut utiliser, en interne, plusieurs processus légers. Supposons que ce soit le cas et que l'objet o d'une activation A soit utilisé par deux processus légers internes à A. Si on applique une des règles permettant l'évolution dynamique des bouquets d'activations que nous avons décrites section 5.5.1 page 153 alors, cet objet o pourrait passer dans une autre activation que A. Dans ce cas, l'activation destinataire pourrait bien être utilisée, par la suite, par les **deux** processus légers qui utilisent o. Le respect du critère d'activabilité ne serait donc plus garanti. C'est pourquoi nous pensons que cette plate-forme ne doit pas permettre la programmation à base de processus légers.

Asynchronisme fonctionnel

Pourtant, l'asynchronisme est parfois une fonctionnalité nécessaire à l'application et dans ce cas il faut qu'il puisse être exprimé par le développeur. C'est le cas par exemple lorsqu'un serveur doit gérer plusieurs connexions simultanément. La syntaxe que nous proposons est asynchrone mais la sémantique est celle d'une exécution séquentielle et aucune garantie ne peut être fournie sur l'exécution asynchrone ou non d'un appel de méthode. Il est donc nécessaire de proposer un mécanisme pour l'asynchronisme explicite. Celui-ci doit permettre au développeur de spécifier que certains appels de méthode doivent être effectués de façon explicitement concurrente, mais également garantir que cette concurrence est conforme au principe des bouquets d'activations. La seule situation

dans laquelle un appel ne doit pas être concurrent est lorsqu'il existe un cycle entre l'appelant et l'appelé puisque dans ce cas, par définition des bouquets d'activations, ceux-ci ne pourront pas être placés dans des activations distinctes. Ils seront forcément dans la même activation et, par conséquent les appels entre eux synchrones. Le compilateur devra donc pouvoir vérifier l'absence de tels cycles afin de garantir que l'appelé pourra être dans une activation distincte de l'appelant. Cette vérification doit être syntaxique afin que le développeur puisse maîtriser facilement ce mécanisme. On peut, par exemple, envisager la définition d'une construction particulière du langage permettant d'effectuer des appels explicitement asynchrones sans aucun moyen d'obtenir une véritable référence sur l'appelé. Cela garanti l'absence de cycle entre appelant et appelé.

Aide à la gestion de l'indépendance des données

L'asynchronisme dans les bouquets d'activations est conditionné par la forme du graphe d'accessibilité et en particulier par le partage des objets dans l'application. Cela garantit la cohérence des données mais peut parfois limiter l'asynchronisme possible inutilement. C'est par exemple le cas dans ces deux situations : l'utilisation par le développeur de références superflues ; le partage de données de grande taille entre objets fonctionnellement indépendants. Dans le premier cas, on peut considérer qu'il s'agit d'une erreur de programmation et il est envisageable de fournir au développeur un outil de type débogueur lui permettant de savoir, pour chaque appel effectué, s'il peut être fait de façon asynchrone ou non et, si non, en raison de quelle dépendance. Cependant, l'effet de certaines des règles que nous avons définies section 5.5.1 page 153 est difficile à prévoir. C'est le cas de la règle 5 page 158 qui peut aboutir à des bouquets d'activations différents selon l'ordre dans lequel elle s'applique. Des règles de ramasse-miettes plus claires et plus systématiques doivent être définies si on veut proposer au développeur ce type de débogueur et si on souhaite qu'il ait la capacité de s'en servir.

Le second cas s'applique par exemple à la modélisation d'une matrice de grande taille par un tableau. Toutes les opérations sur cette matrice partageront ce tableaux et seront donc effectuées de façon séquentielle. Pourtant, les lignes de cette matrice sont indépendantes les unes des autres et les colonnes également. Il serait alors intéressant pour une bibliothèque de ce type de pouvoir déclarer spécifiquement l'absence de partage sur certaines de ses données internes. Une matrice pourrait alors être modélisée comme une liste de lignes et de colonnes, toutes partageant le même tableau mais indiquant explicitement que ni les colonnes ni les lignes ne partagent de données entre elles.

Coût de mise en œuvre

Ces deux éléments évoqués, d'autres questions d'ordre technique sont soulevées par la mise en pratique des bouquets d'activations. Quel est le coût de la gestion des activations, du *déplacement* d'un objet d'une activation vers une autre ? Lorsqu'un objet est déplacé dans une activation, cela modifie la façon dont ses méthodes seront appelées, la ou les

file(s) dans la(les)quelle(s) les appels devront être stockés. Comment ce changement dans le fonctionnement des appels de méthode sur un objet donné peut-il être pris en compte ? Enfin, quel type d'infrastructure faut-il mettre en place pour permettre une gestion efficace des appels asynchrones ? Le mécanisme des références asynchrones proposé au sein de la plate-forme Mandala [135] et en particulier le chaînage de références nous semble être une piste pertinente pour la modélisation d'appels traversant plusieurs activations et pour le développement d'un prototype.

5.6 Conclusion

Dans le cadre de la prise en charge de la latence dans les systèmes distribués, nous nous sommes intéressés ici à la propriété d'activabilité qui permet d'identifier les objets d'un programme qui peuvent être rendus actifs, au sens ProActive du terme, sans modifier la sémantique du programme. Nous avons proposé la propriété d'activabilité étendue section 5.3 page 118. Le respect de cette propriété par un objet dépend autant de l'objet lui-même – ne pas conserver de référence sur les résultats retournés par ses méthodes – que du contexte dans lequel cet objet est utilisé – un seul processus léger appelant et l'appelant ne conserve pas de référence sur les arguments transmis. Nous avons prouvé la validité de cette propriété grâce au π-calcul. Dans le cadre du placement d'objets dans un système distribué, les objets activables peuvent constituer des unités de distribution intéressantes à étudier, les communications entre objets distants devenant asynchrones.

La propriété d'activabilité, même étendue, présente un certain nombre d'inconvénients, en particulier son côté statique. Nous avons, dans un premier temps, tenté de remédier à cela en introduisant la notion de *séparabilité* en 5.4.2 page 148 qui nous permet de faire alterner, durant l'exécution d'un programme, l'état d'un objet entre *activable* et *séparable*. Cependant, même dans ce cas, la possibilité d'introduire de l'asynchronisme dans notre application est directement liée à sa structure.

Les bouquets d'activations lèvent cette limitation : quelque soit la forme du graphe d'accessibilité de l'application, il est toujours possible d'y introduire de l'asynchronisme en le structurant en bouquet (définition 5.14 page 151). La seule limite à cette remarque est le cas des cycles puisque tous les objets d'un même cycle doivent être placés dans la même activation, ce qui évite les problèmes d'étreinte fatale qui pourraient survenir sinon. De plus nous avons vu en 5.5.1 page 153 que la structure des activations peut être mise à jour au fur et à mesure de l'exécution d'une application. Cela évite d'avoir à analyser le programme préalablement, à l'inverse de l'activabilité ou de la séparabilité. Nous avons montré la validité des bouquets d'activations en 5.5.2 page 158 qui découle directement de la validité de l'activabilité étendue.

L'activabilité avait au départ pour objectif de contribuer à la prise en charge de la latence dans les systèmes distribués. L'étude de ce mécanisme nous a conduit à la proposition d'un modèle de programmation concurrente en 5.5.3 page 160. L'analyse plus approfondie et la mise en œuvre de ce nouveau modèle de programmation deviennent des perspectives privilégiées de nos travaux.

Conclusion

Nous nous sommes intéressés dans ce document à cinq caractéristiques des systèmes distribués que nous considérons comme fondamentales. La présentation de ces caractéristiques et l'étude des pistes permettant leur gestion transparente ont fait l'objet du chapitre 1. Nous avons ensuite présenté nos travaux dans le domaine de la prise en charge de la mémoire répartie au chapitre 3 et dans celui de la prise en charge de la latence au chapitre 5. Dans cette conclusion, nous récapitulons l'ensemble de nos contributions puis nous revenons sur la question de l'unification de la programmation locale et distribuée.

Contributions et perspectives

Au delà de ce document de thèse lui-même et de l'éclairage que nous espérons qu'il apporte sur la question de l'unification des paradigmes de programmation locale et distribuée, les travaux menés dans cette thèse constituent des contributions dans divers domaines.

Sécurité des codes mobiles

Nous avons participé à la proposition d'un prototype de grille de calcul sécurisée garantissant une protection du code déployé sur cette grille par l'usage de cartes à puce Java Card. Cette contribution originale a été brièvement présentée section 1.3.2 pages 23 à 26. Nous avons également évoqué les problèmes éthiques soulevés par ce type d'approche pages 26 à 28.

On retrouvera des présentations plus détaillées de la grille de cartes à puce dans [27, 30] ainsi que dans la thèse de Damien Sauveron [114] et le mémoire de DEA d'Achraf Karray [74]. Une grille composée de 32 lecteurs de carte à puces contrôlés par deux serveurs de type PC a été mise en place au LaBRI. Ces travaux se poursuivent actuellement dans le cadre de la thèse d'Achraf Karray.

JToe

Le chapitre 3 nous a permis de présenter l'API JToe que nous avons proposée. La définition de cette API est issue du constat suivant : (1) de nombreux travaux proposent des mécanismes efficaces de transfert d'objets ; (2) ces travaux ne sont pas compatibles entre eux ; (3) ces travaux ne s'intéressent pas *uniquement* au transfert d'objets.

Afin d'unifier les différents travaux existant, il nous a semblé pertinent de proposer une API qui ne s'intéresse qu'au transfert d'objets. Cette API définit un contexte de travail – implantation de l'API – permettant de se concentrer uniquement sur le problème du transfert d'objets. Enfin, cette API permet la mise en œuvre de l'ensemble des optimisations pour le transfert d'objets utilisées dans les travaux existants.

Trois versions de l'API JToe sont actuellement disponibles, deux reposent sur le mécanisme standard de *serialization*, l'une utilisant RMI pour les communications, l'autre directement TCP. La troisième, que nous avons décrite en détails pages 75 à 89, exploite des fonctionnalités spécifiques à la machine virtuelle JikesRVM et transfère directement la représentation mémoire des objets sans copie intermédiaire.

Un certain nombre de perspectives concernant l'implantation de JToe spécifique à JikesRVM ont été énumérées page 89, elles concernent la levée de certaines limitations techniques. L'API JToe elle-même devrait également être améliorée en intégrant la notion de transfert asynchrone qui manque actuellement.

Nous avons déjà décrit cette API et ses implantations dans [28, 29]. Ces implantations sont disponibles sous licence GPL [73]. Les développements sont encore actifs, grâce notamment aux contributions de Benoît Métrot qui a rédigé un mémoire pour sa soumission dans le cadre du Java Community Process afin d'établir un standard dans le domaine du transfert d'objets Java basé sur JToe [97].

L'activabilité

Nous avons proposé la propriété d'activabilité que nous avons décrite au chapitre 5. Cette propriété permet d'ajouter automatiquement de l'asynchronisme dans une application orientée objet sans en modifier la sémantique. Cette propriété constitue une extension de celle proposée dans la thèse de Romain Guider [65]. Dans une application à objets, un objet respectant cette propriété pourra être rendu actif, c'est-à-dire que les appels de méthode sur cet objet deviendront asynchrones et seront traités séquentiellement par un processus dédié. Cette transformation introduira de l'asynchronisme et de la concurrence dans l'application sans en modifier la sémantique.

Afin de prouver formellement la validité de cette propriété, et en particulier le fait que la sémantique de l'application n'est pas modifiée par l'activation des objets qui la respectent, nous avons proposé une modélisation en π-calcul des objets actifs ainsi que des objets activables. Grâce à cela, nous avons pu prouver pages 127 à 145 qu'il existe bien une bisimulation entre ces deux types de processus π-calcul et que, finalement, tout

objet activable peut bien être remplacé par sa version activée, sans modifier la sémantique de l'application.

La propriété d'activabilité constitue un critère intéressant pour le développement d'applications concurrentes. Elle peut constituer un modèle de développement qui permettra au programmeur de s'assurer de la conformité de son application. Mais nous nous sommes également intéressés à la détection automatique des objets activables dans un programme (pages 145 à 148). La brièveté de cette évocation ne reflète pas le temps de travail effectivement consacré à cette question – étude bibliographique, implantation, expérimentation, etc. Cependant, elle reflète parfaitement les résultats obtenus qui n'ont pas été satisfaisants. Le domaine de l'analyse de programmes est un domaine vaste et complexe et qui dépasse largement le cadre de cette thèse. Cependant, notre expérience dans ce cadre nous amène à être sceptique sur l'application de ce type de méthodes à ce problème. Il semble que l'utilisation de méthodes d'analyse statique de programmes risque de demander une compréhension, de la part du développeur, du mécanisme d'analyse mis en jeu afin d'en bénéficier au maximum. De notre point de vue, une telle connaissance est nettement supérieure à ce qui devrait être nécessaire à l'exploitation d'un mécanisme automatique.

Une des limites principales de l'activabilité est le manque de dynamisme puisqu'un objet activable doit demeurer activable tout au long de son existence et ne peut changer d'état. Pour améliorer cela, nous avons introduit, page 148, la propriété de séparabilité qui permet certains changements d'états. Malgré cela, l'activabilité et la séparabilité restent des propriétés statiques qui sont directement liées à la structure des objets du programme. Les bouquets d'activations lèvent cette limite et ouvrent de nouvelles perspectives à nos travaux.

Nous avions déjà présenté une première version de la propriété d'activabilité ainsi que des expérimentations mettant en œuvre des méthodes d'analyse statique de programmes dans [56]. Nous avions prouvé sa validité dans [25]. La preuve présentée dans cette thèse prend en compte la notion de mobilité des objets entre sous-systèmes, notion qui était absente de la publication précédente. Nous avons décrit la notion de séparabilité dans [26] et Christophe Popov a présenté des expérimentations basées sur cette propriété dans son mémoire de DEA [105].

Les bouquets d'activations

L'activabilité permet l'introduction automatique d'asynchronisme dans une application orientée objet sans modification de sa sémantique. Malheureusement, les possibilités offertes, en terme d'asynchronisme, sont directement liées à la structure de l'application. Les bouquets d'activations permettent de lever cette contrainte et offrent une structure beaucoup plus souple. Nous avons décrit les bouquets d'activations à la section 5.5.

Les bouquets d'activations offrent plus de dynamisme que l'activabilité et nous avons décrit un certain nombre de règles pour la mise à jour des bouquets d'activations au fur et à mesure de l'exécution d'un programme qui assurent le respect de la structure en

bouquet de l'application. Ce dynamisme supprime la nécessité d'une analyse préalable du programme, l'asynchronisme pouvant être introduit dynamiquement au fur et à mesure de l'exécution et les objets étant déplacés d'un bouquet à un autre afin de respecter la sémantique séquentielle.

La validité des bouquets d'activations repose directement sur celle de l'activabilité étendue. Le principe est de proposer une transformation d'un programme structuré en bouquet d'activations en un programme normal dans lequel un objet est introduit pour représenter chacune des activations et nous montrons que ce nouvel objet est activable.

Nous décrivons les bouquets d'activations pour la première fois. Nous considérons que l'étude de cette structure constitue une suite privilégiée pour nos travaux. En particulier, nous souhaitons valider de façon plus formelle ce principe. D'autre part, il nous paraît intéressant d'étudier l'ensemble des possibilités en terme de dynamisme dans la structure des bouquets d'activations. Enfin, leur exploitation de façon transparente nous paraît intéressante à étudier également, en particulier le mode d'expression qu'il est nécessaire de mettre en place. Nous avons précisé quelques pistes pour la suite de ces travaux section 5.5.3, notamment la définition d'un langage de programmation dont la syntaxe soit asynchrone – pour permettre une gestion cohérente des exceptions – mais dont la sémantique soit celle d'une exécution séquentielle. Le développement d'autres outils associés est également proposé.

Caractéristiques des systèmes distribués

Le chapitre 1 nous a permis de présenter cinq caractéristiques des systèmes distribués qui nous paraissent essentielles : les pannes partielles, la concurrence, la confiance, la mémoire répartie et la latence. Nous avons présenté différents outils permettant leur prise en charge et nous avons examiné les possibilités de gérer ces caractéristiques de façon transparente.

La section 1.6 page 43 présente nos conclusions sur la gestion transparente de ces caractéristiques. Sans parler d'unification, nous avons constaté une certaine convergence entre paradigme de programmation locale et distribuée résultante de l'évolution des paradigmes de programmation locale et de l'adoption, en leur sein, de mécanismes tels que les processus légers ou encore les modèles de cohérence mémoire relâchés.

Plus généralement, nous avons établi qu'une gestion transparente de ces caractéristiques était possible mais avec un certain nombre de limites : gestion sous-optimale des pannes partielles ; compatibilité entre le modèle mémoire local et un modèle mémoire relâché pour mémoire virtuellement partagée ; confiance dans les machines exploitées pendant l'exécution ; limitation des accès aux ressources de l'application.

Unification des paradigmes de programmation locale et distribuée

En introduction, nous avons défini l'*unification de la programmation locale et distribuée* de façon utopique comme la possibilité de réaliser des programmes selon un paradigme de programmation locale et qu'ils puissent s'exécuter de façon optimale quelque soit le modèle choisi, d'exécution locale ou distribuée. Autrement dit, les caractéristiques spécifiques aux systèmes distribués sont masquées au niveau du langage et leur gestion est reportée au niveau de la plate-forme d'exécution.

L'objectif d'un tel masquage est de fournir une plate-forme qui prenne en charge de façon automatique les caractéristiques d'un système distribué. Ainsi, on peut conserver les applications existantes et les exploiter dans un contexte distribué sans changer ni le code, ni les habitudes de programmation. Cependant, ce type d'approche se heurte rapidement à un certain nombre de limites que nous avons déjà évoqué section 1.6 page 43.

Tant qu'on ne s'intéresse qu'à des applications existantes, ces limitations peuvent être acceptables si le fait même de la distribution apporte un intérêt sur un aspect ou un autre. Dans ce cas, pour un effort de programmation nul, on obtiendra une exécution distribuée de l'application et, pour peu que cette distribution apporte un gain quel qu'il soit, on peut considérer le masquage et sa réalisation comme satisfaisants.

On peut faire le parallèle ici avec les expérimentations que nous avons décrites section 5.4.1 page 145 ayant pour but d'utiliser l'analyse statique de programme pour distribuer une application locale. On obtient alors une application distribuée sans efforts supplémentaires de programmation. Cependant, dans toutes ces expérimentations, l'utilisateur peut intervenir sur les résultats produits par l'analyse afin d'influencer la distribution de l'application. En effet, il parait raisonnable de penser que, rapidement, il va souhaiter modifier son programme de façon à améliorer les résultats fournis par la plate-forme. Dans ce cas, il ne s'agira plus d'une application existante exécutée *gratuitement* de façon distribuée mais bel et bien d'une application prévue pour ou influencée par un modèle d'exécution distribué. Dans ce cas, ce dont aura besoin le développeur, ce n'est plus d'outils qui lui masquent les caractéristiques d'un système distribué mais d'outils qui lui permettent, au contraire, de les prendre en charge le plus facilement possible tout en lui offrant un certain nombre de garanties.

Nous avons vu section 2.2.3 page 60 que certaines plate-formes pouvaient masquer les caractéristiques d'un système distribué – il s'agissait dans ce cas de la mémoire répartie – uniquement sur le plan syntaxique, en laissant à la charge du développeur la prise en charge des conséquences sémantiques de ces caractéristiques. A l'inverse, nous pensons qu'une plate-forme doit, soit totalement masquer au programmeur l'ensemble des caractéristiques d'un système distribué – nous avons présenté des pistes pour cela section 1.6 page 43 – soit lui proposer des outils qui facilitent leur prise en charge explicite et lui permettent d'agir en connaissance de cause.

C’est cette dernière approche qui nous paraît, aujourd’hui, la plus réaliste et vers laquelle s’orientent nos recherches, par exemple avec les bouquets d’activations dans le cadre desquels nous souhaitons proposer une syntaxe explicitement asynchrone.

Bibliographie

[1] Proposition de résolution sur le projet de décision de la Commission constatant le niveau de protection adéquat des données à caractère personnel contenues dans les dossiers des passagers aériens (PNR) transférés au *Bureau des douanes et de la protection des frontières* des États-Unis. Proposition de résolution, Commission des libertés et des droits des citoyens, de la justice et des affaires intérieures du Parlement Européen, 2004.

[2] Chin-Laung Lei Alexander I-Chi Lai. Data prefetching for distributed shared memory systems. In *29th Hawaii International Conference on System Sciences (HICSS'96)*, volume 1 : Software Technology and Architecture, page 102, 1996.

[3] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home : an experiment in public-resource computing. *Commun. ACM*, 45(11) :56–61, 2002.

[4] Gabriel Antoniu. *DSM-PM2 : une plate-forme portable d'implémentation de protocoles de cohérence multithread pour MVP*. PhD thesis, ENS Lyon, novembre 2001.

[5] Gabriel Antoniu, Luc Bougé ;, Philip J. Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. Implementing java consistency using a generic, multithreaded dsm runtime system. In *IPDPS '00 : Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 560–567, London, UK, 2000. Springer-Verlag.

[6] Yariv Aridor, Michael Factor, and Avi Teperman. cjvm : A single system image of a jvm on a cluster. In *ICPP '99 : Proceedings of the 1999 International Conference on Parallel Processing*, page 4, Washington, DC, USA, 1999. IEEE Computer Society.

[7] Yariv Aridor, Michael Factor, and Avi Teperman. Implementing java on clusters. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par*, volume 2150 of *Lecture Notes in Computer Science*, pages 722–731. Springer, 2001.

[8] Isabelle Attali, Denis Caromel, and Romain Guider. Static analysis of java for distributed and parallel programming. Technical Report 3634, INRIA (Institut National de Recherche en Informatique et en Automatique), mars 1999.

[9] Isabelle Attali, Denis Caromel, and Romain Guider. A step toward automatic distribution of java programs. In *FMOODS 2000*, pages 141–161. Kluwer Academic Publishers, 2000.

[10] Françoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. A fault tolerance protocol for ASP calculus : Design and proof. Rapport de recherche RR-5246, INRIA Sophie Antipolis, 2004.

[11] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding communication latency and coherence overhead in software DSMs. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 198–209, 1996.

[12] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3, Bretton Woods, NH, 1983. Association for Computing Machinery.

[13] Fabian Breg and Dennis Gannon. A Customizable Implementation of RMI for High Performance Computing. In *Workshop on Java for Parallel and Distributed Computing of IPPS/SPDP99*, pages 733–747, 1999.

[14] Fabian Breg and Constantine D. Polychronopoulos. Java virtual machine support for object serialization. In *Java Grande*, pages 173–180, 2001.

[15] University of California. Berkeley Open Infrastructure for Network Computing (BOINC). "`http://boinc.berkeley.edu/`" (Dernière visite : 6 avril 2006).

[16] Denis Caromel. Programming Abstractions for Concurrent Programming. In *Technology of Object-Oriented Languages and Systems, PACIFIC (TOOLS PACIFIC '90)*, 1990.

[17] Denis Caromel. *Programmation parallèle asynchrone et impérative : études et propositions*. PhD thesis, Université de Nancy I, 1991.

[18] Denis Caromel and Guillaume Chazarain. Robust exception handling in an asynchronous environment. In Alexander Romanovsky, Dony Christophe, Joergen Lindskov Knudsen, and Anand Tripathi, editors, *Developing Systems that Handle Exceptions*, rapport technique numéro 05-050, LIRMM, Université Montpellier II, France, juillet 2005.

[19] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. *SIGPLAN Not.*, 39(1) :123–134, 2004.

[20] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in java. *Concurrency - Practice and Experience*, 10(11-13) :1043–1061, 1998.

[21] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2) :225–267, 1996.

[22] K. Mani Chandy and Leslie Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[23] Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, and Thorsten von Eicken. Low-latency communication on the IBM RISC System/6000 SP. In *ACM/IEEE Supercomputing '96*, novembre 1996.

[24] P. Chatonnay, B. Herrmann, L. Philippe, and F. Bourdon. Placement dynamique dans les systèmes répartis à objets. *Calculateur parallèle*, 8(1) :11–30, 1996.

[25] Serge Chaumette and Pascal Grange. Parallelizing multithreaded java programs : a criterion and its pi-calculus foundation. In *Workshop on Formal Methods for Parallel Programming/IPDPS*, 2002.

[26] Serge Chaumette and Pascal Grange. Optimizing the execution of a distributed object oriented application by combining static and dynamic information. Poster exposé à International Parallel and Distributed Processing Symposium (IPDPS'2003), 22 - 26 avril 2003.

[27] Serge Chaumette, Pascal Grange, Achraf Karray, Damien Sauveron, and Pierre Vignéras. Secure distributed computing on a Java Card grid. In *Proceedings of the 7th International Workshop on Java for Parallel and Distributed Computing*, Denver, CO, USA, April 48 2005.

[28] Serge Chaumette, Pascal Grange, B. Métrot, and P. Vignéras. JToe : a Java API for object exchange. In Joubert et al. [72], pages 135–142.

[29] Serge Chaumette, Pascal Grange, Benoît Métrot, and Pierre Vignéras. Implementing a high performance object transfert mechanism over jikesrvm. Rapport de recherche 1324-04, Laboratoire Bordelais de Recherche en Informatique (LaBRI), 2004.

[30] Serge Chaumette, Pascal Grange, Damien Sauveron, and Pierre Vignéras. Computing with Java Cards. In *Proceedings of CCCT'03 and 9th ISAS'03*, Orlando, FL, USA, July 31, August 1-2 2003.

[31] Serge Chaumette and Pierre Vignéras. A framework for seamlesly making object oriented applications distributed. In Joubert et al. [72], pages 305–312.

[32] Jonathan J. Cook. Reverse Execution of Java Bytecode. *The Computer Journal*, 45(6) :608–619, 2002.

[33] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems (3rd ed.) : concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[34] L. Courtrai, Y. Mahéo, and F. Raimbault. Expresso : a library for fast java objects transfert. In *Myrinet User Group Conference*, Lyon, 2000.

[35] Patrick Cousot and Radhia Cousot. Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Language*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[36] Culler David, Keeton Kim, Liu Lok Tim, Mainwaring Alan, Martin Rich, Rodrigues Steve, Wright Kristin, and Yoshikawa Chad. Generic Active Message Interface Specification. Technical report, Computer Science Division, University of California at Berkeley, Novembre 1994.

[37] Last Stage of Delirium Research Group. Java and java virtual machine security vulnerabilities and their exploitation techniques, Octobre 2002.

[38] Edsger W. Dijkstra. The structure of the "the"-multiprogramming system. *Commun. ACM*, 11(5) :341–346, 1968.

[39] Samir Djilali, Thomas Herault, Oleg Lodygensky, Tangui Morlier, Gilles Fedak, and Franck Cappello. Rpc-v : Toward fault-tolerant rpc for internet connected desktop grids with volatile nodes. In *SC '04 : Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 39, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Mathias Doreille. *Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. PhD thesis, Institut National Polytechnique de Grenoble, décembre 1999.

[41] Programme Décryphton. "`http://www.decrypthon.fr/`" (Dernière visite : 6 avril 2006).

[42] R. Elz, R. Bush, S. Bradner, and M. Patton. Selection and Operation of Secondary DNS Servers. RFC 2182 (Best Current Practice), July 1997.

[43] ERights.org. "`http://www.erights.org/`" (Dernière visite : 6 avril 2006).

[44] Ted Faison. Interaction patterns for communicating processes. In *Proceedings of Pattern Languages of Programs (PLoP) 98*, 1998.

[45] Weijian Fang, Cho-Li Wang, and Francis C. M. Lau. On the design of global object space for efficient multi-threading java computing on clusters. *Parallel Comput.*, 29(11-12) :1563–1587, 2003.

[46] Adam Ferrari and V. S. Sunderam. Multiparadigm distributed computing with TPVM. *Concurrency : Practice and Experience*, 10(3) :199–228, 1998.

[47] Adam Ferrari and Vaidy S. Sunderam. TPVM : Distributed concurrent computing with lighweight processes. In *HPDC*, pages 211–, 1995.

[48] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[49] Stephany Filimon. Multilevel security : privacy by design. *Crossroads*, 10(3) :4–4, 2004.

[50] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. In *PODS '83 : Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–7, New York, NY, USA, 1983. ACM Press.

[51] The Free Software Foundation. *GNU Lesser General Public License*. "`http://www.gnu.org/copyleft/lesser.html`" (Dernière visite : 6 avril 2006).

[52] G. A. Geist, J. A. Kohla, and P. M. Papadopoulos. PVM and MPI : A Comparison of Features. *Calculateurs Paralleles*, 8(2) :137–150, 1996.

[53] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In David Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 425–440. Springer, 2001.

[54] James Gosling, Bill Joy, and Guy Steele. Threads and Locks. In *The Java Language Specification*, chapter 17. Addison Wesley, 1996.

[55] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *SP '03 : Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 154, Washington, DC, USA, 2003. IEEE Computer Society.

[56] Pascal Grange. Intégration des technologies du distribué à base d'objets dans JEM et distribution automatique d'applications Java. Master's thesis, LaBRI, Université Bordeaux 1, Juin 2000.

[57] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. Rapport technique MSR-TR-2003-96, Microsoft Research, janvier 2004.

[58] Object Management Group. Catalog of omg corba services specifications. "http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm" (Dernière visite : 6 avril 2006).

[59] Object Management Group. Catalog of omg idl / language mappings specifications. "http://www.omg.org/technology/documents/idl2x_spec_catalog.htm" (Dernière visite : 6 avril 2006).

[60] Object Management Group. Idl to java language mapping. version 1.2. août 2002. "http://www.omg.org/cgi-bin/doc?formal/02-08-05" (Dernière visite : 6 avril 2006).

[61] Object Management Group. Object management group. "http://www.omg.org/" (Dernière visite : 6 avril 2006).

[62] Object Management Group. Common object request broker architecture : Core specification. version 3.0.3., mars 2004. "http://www.omg.org/technology/documents/formal/corba_iiop.htm" (Dernière visite : 6 avril 2006).

[63] Object Management Group. Corba core. In *Common Object Request Broker Architecture : Core Specification. Version 3.0.3.* [62], chapter 1–11. "http://www.omg.org/technology/documents/formal/corba_iiop.htm" (Dernière visite : 6 avril 2006).

[64] Object Management Group. Corba messaging. In *Common Object Request Broker Architecture : Core Specification. Version 3.0.3.* [62], chapter 22. "http://www.omg.org/technology/documents/formal/corba_iiop.htm" (Dernière visite : 6 avril 2006).

[65] Romain Guider. *Analyse statique de programmes Java. Application à la parallélisation.* PhD thesis, Université de Nice – Sophia Antipolis, septembre 2000.

[66] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, volume 19, pages 254–265, New York, NY, 1991. ACM Press.

[67] Bernhard Haumacher, Thomas Moschny, and Michael Philippsen. Javaparty. "http://wwwipd.ira.uka.de/JavaParty/" (Dernière visite : 6 avril 2006).

[68] Ludovic Henrio. *Calcul d'objets asynchrone : confluence et déterminisme.* PhD thesis, Université de Nice – Sophie Antipolis, 2003.

[69] C. A. R. Hoare. Monitors : an operating system structuring concept. *Commun. ACM*, 17(10) :549–557, 1974.

[70] IBM. Common public license. "`http://www.eclipse.org/legal/cpl-v10.html`" (Dernière visite : 6 avril 2006).

[71] IBM. *Jikes Research Virtual Machine (RVM)*. "`http://jikesrvm.sourceforge.net`" (Dernière visite : 6 avril 2006).

[72] Gerhard R. Joubert, Wolfgang E. Nagel, F. J. Peters, and W. V. Walter, editors. *Parallel Computing : Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany*, volume 13 of *Advances in Parallel Computing*. Elsevier, 2004.

[73] The JToe project. "`http://jtoe.sf.net`" (Dernière visite : 6 avril 2006).

[74] Achraf Karray. Calcul sécurisé sur grille de cartes à puce. Master's thesis, École Nationale d'Ingéniereurs de Sfax, juin 2004.

[75] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA '92 : Proceedings of the 19th annual international symposium on Computer architecture*, pages 13–21, New York, NY, USA, 1992. ACM Press.

[76] K. Kono and T. Masuda. Efficient rmi : Dynamic specialization of object serialization. In *Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 308–315, 2000.

[77] Markus Kuhn. The trustno 1 cryptoprocessor concept. Rapport de recherche CS555, Purdue University, 1997.

[78] Dawid Kurzyniec, Tomasz Wrzosek, Vaidy Sunderam, and Slominski Aleksander. Rmix : Multiprotocol rmi framework for java. In *Joint ACM Java Grande – ISCOPE 2002 Conference*, novembre 2002.

[79] Dawid Kurzyniec, Tomasz Wrzosek, and Vaidy S. Sunderam. Heterogeneous access to service-based distributed computing : The rmix approach. In *Heterogeneous Computing Workshop (HCW), 17th International Parallel and Distributed Processing Symposium (IPDPS-2003)*, Nice, France. IEEE Computer Society.

[80] Dawid Kurzyniec, Tomasz Wrzosek, Vaidy S. Sunderam, and Aleksander Slominski. RMIX : A multiprotocol RMI framework for java. In *Workshop on Java for Parallel and Distributed Computing (JAVAPDC), 17th International Parallel and Distributed Processing Symposium (IPDPS-2003)*, Nice, France. IEEE Computer Society.

[81] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, 1978.

[82] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4) :18–25, 2001.

[83] Pascale Launay and Jean-Louis Pazat. Ecrire parallèle, exécuter distribué. *Technique et science informatiques*, 19(9) :1193 – 1221, novembre 2000.

[84] R. Greg Lavender and Douglas C. Schmidt. Active object : an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, 1995.

[85] Tim Lindholm and William Pugh. JSR 133 : Java Memory Model and Thread Specification Revision. Technical report, Java Community Process, septembre 2004.

[86] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3) :300–312, 1988.

[87] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel J. H. Jacobs, and Rutger F. H. Hofman. Efficient java RMI for parallel programming. *Programming Languages and Systems*, 23(6) :747–775, 2001.

[88] Microsoft. Microsoft .net homepage. "`http://www.microsoft.com/net/`" (Dernière visite : 6 avril 2006).

[89] Sun Microsystem. Java technology. "`http://java.sun.com/`" (Dernière visite : 6 avril 2006).

[90] Sun Microsystem. Javabeans. "`http://java.sun.com/products/javabeans/docs/beans.101.pdf`" (Dernière visite : 6 avril 2006).

[91] Sun Microsystems. Java object serialization specification, 2003. "`http://java.sun.com/j2se/1.4.2/docs/guide/serialization/spec/serialTOC.html`" (Dernière visite : 6 avril 2006).

[92] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency Among Strangers : Programming in E as Plan Coordination. In *TGC 2005*, volume 3705 of *Lecture Notes in Computer Science*, pages 195 – 229, 2005.

[93] Robin Milner. *Communicating and mobile systems : the π-calculus*. Cambridge University Press, Cambridge, UK, 1999.

[94] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035.

[95] Todd C. Mowry, Charles Q. C. Chan, and Adley K. W. Lo. Comparative evaluation of latency tolerance techniques for software distributed shared memory. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pages 300–311, 1998.

[96] Myricom. Myrinet software. "`http://www.myri.com/scs/`" (Dernière visite : 6 avril 2006).

[97] Benoit Métrot. Etablissement d'un standard pour une plate-forme canonique de transfert d'objet. Master's thesis, Université Bordeaux 1 - Département d'informatique, Juin 2005.

[98] Mohamed Naimi, Michel Trehel, and André Arnold. A log (n) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1) :1–13, 1996.

[99] R. Namyst. *PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Université de Lille 1, Janvier 1997.

[100] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for java. In *Java Grande*, pages 152–159, 1999.

[101] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[102] The Mozilla Organization. Link prefetching faq. "`http://www.mozilla.org/projects/netlib/Link_Prefetching_FAQ.html`" (Dernière visite : 6 avril 2006).

[103] Michael Pilippsen and Matthias Zenger. Javaparty - transparent remote objects in java. In *Concurrency : practice and experience*, volume 9, pages 1225–1242, 1997.

[104] Raquel Pinto, Ricardo Bianchini, and Claudio L. Amorin. Comparing latency-tolerance techniques for software dsm systems. *Parallel and Distributed Systems, IEEE Transactions on*, 14 :1180–1190, novembre 2003.

[105] Christophe Popov. Raffinement dynamique d'informations statiques pour la répartition dans les systèmes distribués à objets. Master's thesis, LaBRI, Université Bordeaux 1, juin 2003.

[106] ProActive. "`http://proactive.objectweb.org/`" (Dernière visite : 6 avril 2006).

[107] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25(2) :47–50, 1991.

[108] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the java system. *Computing Systems*, 9(4) :291–312, 1996.

[109] David Safford. Clarifying misinformation on tcpa. "`http://www.research.ibm.com/gsal/tcpa/tcpa_rebuttal.pdf`" (Dernière visite : 6 avril 2006), Octobre 2002.

[110] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Langõages and Systems*, 20(1) :1–50, January 1998.

[111] David Sagnol. π-calcul pour les langages à objets parallèles. Master's thesis, Université de Nice Sophia-Antipolis, 1995. Mémoire de DEA.

[112] Bo I. Sandén. Entity-life modeling : Modeling a thread architecture on the problem environment. *IEEE Softw.*, 20(4) :70–78, 2003.

[113] Luis F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Gener. Comput. Syst.*, 18(4) :561–572, 2002.

[114] Damien Sauveron. *Étude et réalisation d'un environnement d'expérimentation et de modélisation pour la technologie Java Card. Application à la sécurité*. PhD thesis, Université Bordeaux 1, 2004.

[115] G. Shah, J. Nieplocha, C. Mirza, R. Harrison, R.K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with lapi : a new high-performance communication library for the ibm rs/6000 sp. In *International Parallel Processing Symposium*, pages 260–266, 1998.

[116] André Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, Freie UniversitÄat Berlin, 2002.

[117] André Spiegel. Distributed Computing : A Note on A Note. In *Automatic Distribution of Object-Oriented Programs* [116], pages 50 – 53.

[118] Team Squid. Squid web proxy cache. "`http://www.squid-cache.org/`" (Dernière visite : 6 avril 2006).

[119] R. Srinivasan. RPC : Remote Procedure Call Protocol Specification Version 2. RFC 1831 (Proposed Standard), August 1995.

[120] Sun Microsystem. *Java API for XML-Based RPC (JAX-RPC)*. "`http://java.sun.com/xml/jaxrpc/`" (Dernière visite : 6 avril 2006).

[121] Sun Microsystem. *Java Authentication and Authorization Service (JAAS)*. "`http://java.sun.com/products/jaas/`" (Dernière visite : 6 avril 2006).

[122] Sun Microsystem. *Java RMI over IIOP*. "`http://java.sun.com/products/rmi-iiop/`" (Dernière visite : 6 avril 2006).

[123] Sun Microsystem. *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated ?* "`http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html`" (Dernière visite : 6 avril 2006).

[124] Mihai Surdeanu and Dan Moldovan. Design and performance analysis of a distributed java virtual machine. *IEEE Trans. Parallel Distrib. Syst.*, 13(6) :611–627, 2002.

[125] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems : Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[126] The kaffe Projet. "`http://www.kaffe.org/`" (Dernière visite : 6 avril 2006).

[127] The Mandala Project. "`http://mandala.sourceforge.net/`" (Dernière visite : 6 avril 2006).

[128] The Open Group. The Single UNIX Specification, Version 2. Page web présentant le manuel de `pthread_cancel`. "`http://www.opengroup.org/onlinepubs/007908799/xsh/pthread_cancel.html`" (Dernière visite : 6 avril 2006).

[129] Kritchalach Thitikamol and Peter J. Keleher. Multi-threading and remote latency in software DSMs. In *Proceedings, 14th International Conference on Distributed Computing Systems*, pages 296–304, May 1997.

[130] Theo Ungerer, Borut Robic, and Jurij Silc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1) :29–63, 2003.

[131] Leendert van Dorn and Andrew S. Tanenbaum. Using active messages to support shared objects. In *Proceedings of the 6th workshop on ACM SIGOPS European workshop*, pages 112–116. ACM Press, 1994.

[132] Rob Van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Ibis : an efficient java-based grid programming environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, 2002.

[133] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Runtime optimizations for a java dsm implementation. In *JGI '01 : Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 153–162, New York, NY, USA, 2001. ACM Press.

[134] Ronald Veldema and Henri E. Bal. Optimizing java-specific overheads : Java at the speed of c ? In European *High Performance Computing & Networking (HPCN)*, juin 2001.

[135] Pierre Vignéras. *Vers une programmation locale et distribuée unifiée au travers de l'utilisation de conteneurs actifs et de références asynchrones.* PhD thesis, Université Bordeaux 1, 2004.

[136] Werner Vogels, Robbert van Renesse, and Ken Birman. Six misconceptions about reliable distributed computing. In *EW 8 : Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 276–279, New York, NY, USA, 1998. ACM Press.

[137] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages : A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[138] Thorsten von Eicken and Werner Vogels. Evolution of the virtual interface architecture. *Computer*, 31(11) :61–68, 1998.

[139] The World Wide Web Consortium W3C. Basic html data types. "`http://www.w3.org/TR/html4/types.html`" (Dernière visite : 6 avril 2006).

[140] The World Wide Web Consortium W3C. *SOAP Specification*. "`http://www.w3.org/TR/soap/`" (Dernière visite : 6 avril 2006).

[141] Jim Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6(3) :5–7, 1998.

[142] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7) :76–82, 1999.

[143] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *MOS '96 : Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64, London, UK, 1997. Springer-Verlag.

[144] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10) :187–206, 1999.

[145] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the java system. In *Second USENIX Conference on Object-Oriented Technologies (COOTS)*, 1996.

[146] Tomasz Wrzosek, Dawid Kurzyniec, and Vaidy S. Sunderam. Performance and client heterogeneity in service-based metacomputing. In *Heterogeneous Computing Workshop (HCW), 18th International Parallel and Distributed Processing Symposium (IPDPS-2004)*, Santa Fe, New Mexico, USA. IEEE Computer Society.

[147] John N. Zigman and Ramesh Sankaranarayana. djvm - a distributed jvm on a cluster. Technical Report TR-CS-02-04, Department of Computer Science, FEIT, ANU, 2002.

Index

Symboles

A