Système d'agents mobiles pour les architectures de calculs auto-adaptatifs

Cyril Dumont

UNIVERSITÉ PARIS-EST

Université Paris-Est

École Doctorale MSTIC

Système d'agents mobiles pour les architectures de calculs auto-adaptatifs

THÈSE

présentée et soutenue publiquement le **28 mai 2014**

en vue de l'obtention du grade de

Docteur de l'Université Paris-Est

(Spécialité Informatique)

par

Cyril DUMONT

Composition du jury

Rapporteurs :	Fabienne Boyer	LIG - Université Joseph Fourier (Maître de conférences HDR)
	Jean-Paul Bodeveix	IRIT - Université de Toulouse III (Professeur)
Examinateurs :	Samia Bouzefrane	CEDRIC - CNAM (Maître de conférences HDR)
	Gilles Roussel	LIGM - Université Paris-Est (Professeur)
	Catalin Dima	LACL - Université Paris-Est (Professeur)
Directeur :	Fabrice Mourlin	LACL - Université Paris-Est (Maître de conférences HDR)

Mis en page avec la classe thloria.

Remerciements

En premier lieu, je voudrais remercier Fabienne Boyer et Jean-Paul Bodeveix d'avoir accepté de relire cette thèse et d'en être les rapporteurs. Leurs remarques et les échanges que nous avons pu avoir ont été très précieux dans la finalisation de ce travail. Merci également aux autres membres du jury, Samia Bouzefrane, Catalin Dima et Gilles Roussel, d'avoir participé à la soutenance de cette thèse et à la pertinence de leurs remarques et de leurs questions.

Bien entendu, je remercie tout particulièrement mon directeur de thèse, Fabrice Mourlin, qui a toujours été disponible durant ces nombreuses années. A l'écoute de mes nombreuses questions, très attentif à l'avancée de mes travaux, cette thèse lui doit vraiment beaucoup. Pour tout cela merci.

Je remercie aussi Laurent Nel et Leuville Objects sans qui je n'aurais pas pu commencer cette thèse. Merci également à Chantal Reynaud et Jean-Christophe Souplet qui ont m'ont accueilli au LRI alors que ma thèse était sur le point d'être achevée et qui ont toujours accepté de me laisser du temps pour la finir.

Je remercie également Emmanuel et Julie Fournet qui m'ont permis d'éviter d'être ridicule dans la langue de Shakespeare...

Merci aussi à toutes les personnes qui, durant toutes ces années, m'ont posé la question "Et ta thèse ... tu en es où ?". Même si parfois cela n'était pas très agréable à entendre, cela m'a le plus souvent remotivé et donné envie de la finir pour pouvoir juste répondre "Ça y est, c'est fini !".

Et surtout, je remercie Clémence qui a su me soutenir, me supporter, m'encourager, me remotiver... pendant toute la durée de ma thèse et qui a su trouver les mots pour expliquer ce travail aux non-informaticiens. Et merci enfin à ma petite Séraphine pour avoir aidé sa mère à supporter mon absence pendant les nombreux week-ends nécessaires à la préparation de ma soutenance...

À Clémence et Séraphine

Table des matières

Table des figures

Liste des tableaux

Introduction

Depuis très longtemps, les scientifiques utilisent la capacité des ordinateurs dans le but de simuler numériquement des phénomènes physiques complexes. Grâce à cela, ils réussissent à répondre à des questions non accessibles par des voies purement expérimentales ou théoriques. Ce domaine de recherche, qu'est la simulation numérique, permet de mieux comprendre ces phénomènes et d'affiner leur modélisation. Il est possible de définir de manière générale la simulation numérique comme étant le processus qui permet de reproduire numériquement sur ordinateur un phénomène décrit par un ou plusieurs modèles mathématiques. En d'autres termes, la simulation numérique est une approche qui permet aux chercheurs d'analyser des phénomènes qui par leur complexité échappent au calcul traditionnel.

Motivations et contributions

Le calcul numérique est un domaine riche en contraintes de toutes natures. La gestion des données peut respecter des schémas précis pour assurer leur disponibilité ou leur diffusion. Le temps d'exécution doit appartenir à une fenêtre temporelle précise. Les ressources utilisées pour une simulation apportent elles aussi leur lot de contraintes. Ainsi, une topologie de processeurs, respectant une configuration demandée, sera réservée pour une durée fixée avec des services pré-déployés. Cet aperçu n'est que le reflet du quotidien d'un numéricien. Il est certain que la préparation d'une campagne de simulations nécessite de fournir un ensemble de paramètres afin que cette campagne puisse être lancée. Cette recherche de la meilleure configuration est essentielle mais n'est pas suffisante pour assurer que les cas de calcul puissent se dérouler dans de bonnes conditions. Un incident peut survenir et ainsi mettre en échec toute la préparation.

Notre expérience nous conduit souvent à relater des incidents survenus dans des campagnes passées qui ont coûté la perte d'heures de calcul. Ainsi, la lecture de données non valides, ne respectant pas un format convenu peut amener une levée d'exception au plus tôt d'une simulation. L'impossibilité d'accéder à un port de communication convenu peut lever un signal qui ne sera pas traité et fera échouer le calcul. Plus fréquent encore est l'absence de ressource de calcul disponible, le calcul attend qu'une ressource se libère ; comme cela n'arrive pas le cas de calcul attend sans activité jusqu'à la fin de la réservation de la configuration. Aucun résultat n'est

obtenu. Seules des observations continues peuvent permettre de réagir au plus tôt dans ce cas de figure.

Les observations d'un cas de calcul sont généralement obtenues par des plates-formes support d'exécution du cas de calcul. Les unes portent sur l'activité des processeurs, ainsi, l'absence d'activité est une alerte évidente. Les autres sont plus intrusives car en rapport avec le cas de calcul en cours. Ainsi, un taux de convergence peut être affiché, un nombre d'itérations dans un calcul différentiel. Bref, ce sont des métriques qui ne sont conçues que par les auteurs du cas de calcul lui-même. Dans la situation où la bonne métrique est observée, le déclenchement de la bonne action est encore plus hypothétique. Ainsi, le choix d'un autre port de communication doit être mis en place dans la configuration du cas de calcul. Lorsqu'un nombre d'itérations dépasse un seuil prévu, il est plus utile de stopper la simulation pour faire une analyse des données contextuelles.

Étant extérieur à l'évaluation du cas de calcul, l'administrateur de la plate-forme numérique n'est pas nécessairement le plus à même pour intervenir rapidement sur un incident en rapport avec le problème à traiter. Ainsi, il apparaît naturel que la gestion de l'incident doit venir de la plate-forme elle-même. La reconfiguration d'un port de communication semble facile à automatiser, en testant le premier port libre d'une plage. Le dépassement d'un nombre d'itérations convenu peut lui aussi être arrêté dès le seuil franchi avec la possibilité de recommencer ce calcul afin de déterminer si l'incident est déterministe ou plus simplement de sauver sur un support persistant les informations contextuelles. L'état du calcul en anomalie peut aussi être notifié dans un journal.

Contenu de la thèse

L'organisation du document est la suivante :

- Dans le chapitre 1 nous présentons le contexte de recherche dans lequel se situe cette thèse. Nous y abordons les notions de calcul distribué, d'adaptabilité, de tolérance aux pannes et de mobilité avec les agents mobiles.
- Dans le chapitre 2 nous définissons formellement une architecture logicielle d'un système capable de s'adapter à son environnement d'exécution dans le but de résoudre des cas de calcul numérique. Nous commençons par décrire le π-calcul polyadique d'ordre supérieur avec lequel nous définissons l'architecture.
- Dans le chapitre 3 nous transformons cette architecture logicielle en un réseau d'automates temporisées pour en vérifier des propriétés temporelles à l'aide de l'outil de *model-checking* UppAal. Pour cela, nous définissons un certain nombre d'opérateurs et de règles de transformation que nous appliquons sur les termes définis en π-calcul au chapitre précédent.
- Dans le chapitre 4 nous décrivons notre framework *MCA* qui implante l'architecture définie précédemment. Nous listons les différents composants qui communiquent pour la résolution d'un cas de calcul, en particulier les *spaces* qui proposent une mémoire partagée.

– Dans le chapitre 5 nous évaluons notre framework *MCA*. Les différentes expérimentations que nous réalisons permettent de valider l'adaptabilité, la tolérance aux pannes et la facilité de développement offertes par notre framework *MCA*.
– Pour finir, nous concluons ce document par un bilan de nos contributions et nous présentons les perspectives ouvertes par notre travail.

A la fin du document figurent la liste de références utilisées dans nos cinq chapitres.

Chapitre 1

Contexte de recherche

Sommaire

L'évolution des capacités de traitement des stations de travail et l'amélioration de la qualité du réseau reliant ces stations a fait évoluer le calcul distribué au cours de ces dernières années. Internet permet à des millions de machines de communiquer entre elles via son réseau IP et propose un environnement réparti dans lequel des programmes peuvent être exécutés en parallèle. Aujourd'hui, une machine connectée au réseau Internet est en état de repos la plupart de son temps et une bonne utilisation de ses ressources permet de profiter de sa puissance sans affecter ses propres travaux. Ainsi, cette plate-forme virtuelle permet de créer une puissance de calcul potentiellement « sans limite ».

L'objectif de ce chapitre est de présenter le contexte de recherche dans lequel se situe cette thèse. Les notions abordées constituent les bases de notre travail mais ne sont pas exhaustives. La première section de ce chapitre introduit le domaine du calcul distribué et en présente quelques exemples d'architectures existantes. Ensuite, les sections 1.2 et 1.3 se concentrent sur deux caractéristiques essentielles qui rendent un système distribué performant : l'adaptabilité et la tolérance aux pannes. Le section 1.4 définit la notion de mobilité dans le domaine du calcul distribué, montre comment elle peut être utilisée pour répondre aux exigences d'un système distribué et énumère les problèmes de sécurité intrinsèques à ce type d'architecture. Enfin, la section 1.5 conclut ce chapitre par un bilan des besoins lors de la mise en place d'une application distribuée.

1.1 Architecture logicielle pour le calcul distribué

Le terme de « grille » (*grid* en anglais) a été répandu en 1998 par Ian Foster et Carl Kesselman [FK98]. Dans cet ouvrage, les auteurs expliquent qu'ils ont choisi le terme de « grille » par analogie avec le réseau de distribution électrique américain, car une grille informatique fournit de la puissance (de calcul ou de stockage de données) de façon *transparente* comme le réseau de distribution électrique fournit de la puissance électrique.

Dans [FK98] il est donné la définition suivante :

> « A computational grid is a hardware and software infrastructure that provide dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities. »

Une grille de calcul est donc une organisation virtuelle qui exploite la puissance de calcul d'un nombre très important de machines hétérogènes (pouvant aller jusqu'à plusieurs milliers) et permet de réaliser des calculs distribués.

1.1.1 Architecture logicielle

La conception d'applications pour ce type de système est complexe. Les applications sont composées de processus distribués géographiquement qui s'exécutent sur les nœuds appartenant aux grilles. Lors de l'exécution d'une application distribuée, les processus qui la composent coopèrent notamment en partageant des données. L'IEEE donne la définition suivante (IEEE Std 1471-2000) :

> « Une architecture logicielle est l'organisation fondamentale d'un système incarnée dans ses composants, leurs relations avec chacun des autres et avec l'environnement, et les principes guidant sa conception et son évolution. »

A partir de cette définition, nous pouvons dire que la description architecturale d'un système spécifie :

– sa structure : les différents composants présents,
– son comportement : les interactions entre les différents composants ainsi que les protocoles de communications utilisés,
– ses propriétés globales : fonctionnelles et non fonctionnelles.

1.1.2 Exigences non fonctionnelles

Les exigences fonctionnelles décrivent les fonctionnalités que le système doit remplir, ou celles de chacun des composants du système. Les exigences fonctionnelles explicitent le comportement attendu du système, ainsi que ses entrées et ses sorties. De leur coté, les exigences non fonctionnelles servent à spécifier les caractéristiques du système, ses atouts.

Des études ont été menées pour normaliser ces concepts. La norme *RM-ODP*(*Reference Model of Open Distributed Processing*) [ISO09] a donné un cadre pour la spécification de systèmes distribués. Nous ne rentrons pas dans les détails de cette norme mais il nous a semblé intéressant de présenter la notion de transparence, qui est la capacité de cacher à l'utilisateur les aspects techniques et organisationnels d'un système distribué. Cela permet à un utilisateur de bénéficier des services offerts par le système sans en connaître les détails techniques et la localisation des ressources qui fournissent ces services. La norme *RM-ODP* définit plusieurs types de transparences :

– *transparence d'accès* : l'accès (local ou distant) à une ressource ou le format d'une donnée sont cachés à l'utilisateur.
– *transparence de localisation* : l'utilisateur ne connaît pas la localisation de la ressource qu'il utilise [LBCC08].
– *transparence de concurrence* : l'exécution de plusieurs processus en parallèle ou le partage d'une ressource par plusieurs utilisateurs sont cachés.
– *transparence de réplication* : l'utilisateur ne sait pas que certaines ressources ou données sont dupliquées pour rendre le système plus fiable.
– *transparence de mobilité* : une ressource peut se déplacer sur le réseau sans en informer directement l'utilisateur.
– *transparence de panne* : la panne d'une ou plusieurs ressources du système doit être invisible à l'utilisateur.
– *transparence de performance* : le système peut être reconfiguré lors de son exécution pour augmenter ses performances sans demander l'intervention de l'utilisateur.
– *transparence d'échelle* : l'augmentation du nombre de ressources ou de la demande de calcul ne doit pas détériorer la performance du système.

Une des qualités essentielles d'un système distribué est la capacité à se rapprocher le plus possible d'une transparence totale : plus celle-ci est proche, plus le système est vu comme une application unique et cohérente. A partir de la liste de ces transparences, il est possible de définir trois exigences non fonctionnelles d'un système distribué :

1.1.2.1 Passage à l'échelle

Le concept de passage à l'échelle désigne la capacité d'un système à continuer à délivrer un service avec un temps de réponse constant même si le nombre de clients ou de données augmente de manière importante.

1.1.2.2 Disponibilité

La disponibilité d'un système est sa capacité à délivrer correctement le ou les services de manière conforme à sa spécification. Pour rendre un système disponible, il faut donc le rendre capable de répondre à tout problème qui peut compromettre son bon fonctionnement.

L'indisponibilité d'un système peut être causée par plusieurs sources parmi lesquelles nous pouvons citer :

- les pannes qui sont des conditions ou évènements accidentels empêchant le système, ou un de ses composants, de fonctionner de manière conforme à sa spécification ;
- les surcharges qui sont des sollicitations excessives d'une ressource du système entraînant sa congestion et la dégradation des performances du système ;
- les attaques de sécurité qui sont des tentatives délibérées pour perturber le bon fonctionnement du système, engendrant des pertes de données et de cohérences ou l'arrêt du système.

1.1.2.3 Autonomie

Un système ou un composant est dit autonome si son fonctionnement ou son intégration dans un système existant ne nécessite aucune modification des composants du système hôte. L'autonomie des composants d'un système favorise l'adaptabilité, l'extensibilité et la réutilisation des ressources de ce système.

1.1.3 Modèles de programmation

Une application distribuée est constituée d'un ensemble de programmes, répartis sur plusieurs machines, qui exécutent ou non le même code sur des données identiques ou non.

Rendre un algorithme parallèle consiste, le plus souvent, à découper les données sur lesquelles il travaille et à définir une tâche pour chaque partie des données. Chaque tâche est alors affectée à une machine particulière. Bien entendu, sauf dans le cas où les calculs d'une tâche ne dépendent pas des autres tâches, il faut mettre en œuvre des communications entre les machines exécutant des tâches dépendantes. On peut donc décomposer ces algorithmes en étapes de calculs et en étapes de communications. L'objectif recherché dans la distribution d'un algorithme étant d'accélérer sa vitesse d'exécution, nous aurons constamment à vérifier deux exigences souvent

contradictoires : minimiser la communication et équilibrer au mieux la charge de travail des machines disponibles. Le cycle de vie d'une application distribuée typique peut se décomposer de la manière suivante :

1. Initialisation du système à partir des paramètres fournis par l'utilisateur.
2. Prétraitement des données.
3. Répartition du calcul sur un maximum de machines disponibles.
4. Exécution des traitements en parallèle par chaque machine.
5. Collecte des résultats et post-traitement pour le calcul du résultat final.

L'approche la plus simple pour créer une application distribuée est le paradigme « *Maître-Travailleurs* ». Celui-ci consiste à créer un programme « maître » qui déclenche d'autres programmes, dit « travailleurs », et attend la fin de leur exécution pour récupérer leurs résultats. Le processus exécutant le programme *maître* traite alors les résultats au cas par cas, ou les agrège, pour à nouveau lancer des sous-programmes (*travailleurs*) ou alors finaliser le calcul.

Dans la suite, nous présentons différents modèles de programmation pouvant être utilisés pour mettre en place une application distribuée sur une grille de calcul. Pour chaque modèle, nous présentons un exemple d'implémentation. Cette liste n'est pas exhaustive mais donne une vue globale des possibilités d'utilisation des ressources d'une grille de calcul.

1.1.3.1 Échanges de messages

Le modèle *SPMD* permet d'organiser l'exécution d'un cas de calcul sur un ensemble de processus comme le propose les grilles de calcul. Dans ce modèle, un programme unique est chargé sur chaque nœud de calcul. Chaque copie du programme est exécutée indépendamment et se synchronise avec les autres processus à l'aide de messages. Chaque processus est identifié par un numéro de rang. Cet identifiant unique est utilisé, par exemple, afin de connaître sur quelle partie des données est exécuté le programme.

La grande majorité des codes scientifiques sont des applications parallèles utilisant des bibliothèques d'échanges de messages pour communiquer. Dans ce modèle, plusieurs processus travaillent sur des données locales. Chaque processus a ses propres variables et n'a pas accès directement aux variables des autres processus. Pour échanger des données, les processus s'envoient des messages. *MPI* (*Message Passing Interface*) [Pac97] est considéré comme le standard de la programmation parallèle par échange de messages. Une application *MPI* est composée d'un nombre de processus fixé au départ (du moins pour *MPI-1*), communiquant par échange de messages, exécutant un ou plusieurs programmes en parallèle (*SPMD* ou *MPMD*[1]). *MPI* permet de communiquer en point-à-point (`send()`/`receive()`) ou via des opérations collectives (`gather()`, `broadcast()`, etc...) et offre différents modes de dialogue (synchrone/asynchrone, bloquant ou non, avec tampon ou sans). *MPI* n'étant qu'un standard, il en existe de nombreuses implantations

1. *SPMD* pour *Simple Program Multiple Data* et *MPMD* pour *Multiple Program Multiple Data*

dont la plupart sont fournies par les constructeurs de machines parallèles. Ces implantations sont optimisées pour un type de machine particulière, mais sont rarement interopérables. Notons que la version *MPI-2* [GLT99] permet dans une certaine mesure la connexion dynamique de plusieurs programmes *MPI* selon un modèle client/serveur et l'ajout dynamique de processus au sein d'un communicateur (*spawn*).

1.1.3.2 Appels de méthodes à distance

L'utilisation de langages objet, comme *Java* ou *C++*, permet de réaliser des appels de méthodes à distance. L'emploi d'une approche objet permet de réduire les erreurs grâce notamment à la gestion des exceptions. Les communications entre les services offerts par des instances de classes et les clients sont souvent assurées par un *bus logiciel*. Un *bus logiciel* (ou *middleware*) offre les propriétés suivantes :

- *La séparation des interfaces et de l'implémentation des objets* : ce qui permet aux concepteurs d'applications d'avoir accès à des services sans se préoccuper de leurs implémentations.
- *La transparence au protocole de communication* : la conception de l'application est indépendante de la technologie de communication utilisée.
- *La transparence à la localisation des objets* : grâce à un service de nommage, les objets distants sont accessibles sans avoir à connaître leur localisation sur le réseau.

Salome

CORBA (*Common Object Request Broker Architecture*) [Obj04] est une spécification de bus logiciel standardisée par l'*OMG*. L'*ORB*, au centre de cette architecture, assure l'automatisation des tâches de communication, de localisation, d'activation d'objets, et de la transmission des messages échangés entre systèmes hétérogènes.

SALOME est une plate-forme logicielle dédiée à la réalisation et au pilotage d'applications de simulation numérique. La plate-forme SALOME est issue d'un co-développement entre plusieurs partenaires dont EDF R&D, CEA et Open CASCADE. SALOME est une plate-forme open source générique qui propose des outils de pré/post-traitement et de couplage de codes de calcul pour la simulation numérique [Dav06]. Le noyau de SALOME est fondé sur la technologie CORBA.

1.1.3.3 Les conteneurs d'*objets actifs*

Il existe peu d'environnement numérique permettant d'exploiter des conteneurs d'objets. *Proactive* [BBC+06] est une référence de cette approche et offre un environnement de développement fourni sous la forme d'une bibliothèque Java. Cette bibliothèque facilite la mise en place de calcul parallèle, distribué et concurrent. L'utilisation de la machine virtuelle Java permet une grande

souplesse vis-à-vis des nombreuses architectures (matérielles ou logicielles) qui peuvent être présentes sur une grille de calcul. Aucune modification de l'environnement Java n'est requise et une application développée avec la librairie *ProActive* s'exécute avec la machine virtuelle fournie par *Oracle*.

Les objets actifs

La plate-forme *ProActive* repose sur la notion d'objet *actif*. Un objet *actif* est un objet Java auquel on a rajouté des comportements supplémentaires. La transparence de sa localisation physique et la gestion de la synchronisation en sont des exemples. Une application développée avec la librairie *ProActive* est structurée en *sous-systèmes*. Un *sous-système* est composé d'un objet *actif* et d'un graphe d'objets *passifs*. Il n'y a qu'un seul objet actif par sous-système. Un objet actif possède son propre *thread*, ce qui lui permet de répondre de façon asynchrone aux différents appels de méthodes qui lui sont destinés. Un objet passif ne peut pas avoir de référence vers un autre objet passif appartenant à un sous-système autre que le sien mais peut avoir une référence vers tous les objets actifs. La figure 1.1 illustre les différents types de communication possibles dans une application *ProActive*.

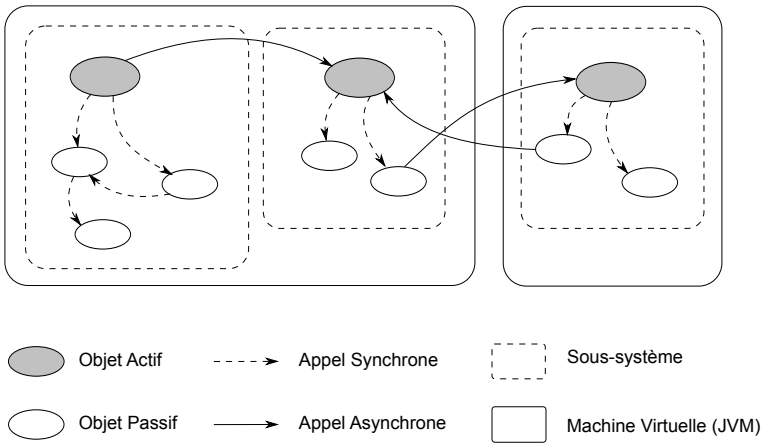


FIGURE 1.1 – Communications entre objets dans une application *ProActive*

L'appel d'une méthode sur un objet actif est asynchrone : quand une méthode est appelée sur un objet actif, elle renvoie instantanément un objet *future* qui est un espace réservé pour accueillir le résultat de l'invocation de la méthode. Pour l'appelant, il n'y a aucune différence entre cet objet *future* et un objet que pourrait retourner la même méthode appelée sur un objet non actif. L'appelant peut ainsi continuer son exécution comme si la méthode avait été appelée de manière synchrone. Le rôle de l'objet *future* est de bloquer le thread dans lequel il s'exécute tant qu'il n'a pas reçu le résultat de l'objet actif. Pendant ce temps, l'appelant peut appeler des méthodes sur l'objet *future*, ce dernier les exécutera quand son thread sera débloqué. Cette approche fournit une forme de transparence d'accès.

Migration

La bibliothèque logicielle *ProActive* offre la possibilité de faire migrer des objets actifs d'un site vers un autre. La migration consiste à sérialiser tout le sous-système de l'objet actif (lui et ses objets passifs associés) et de le reconstruire à son arrivée sur le site distant. L'agent actif lui-même prend la décision de migrer vers un site distant. Pour assurer les communications vers l'agent actif après ses différentes migrations, il est important de pouvoir le localiser. Pour cela, *ProActive* utilise la notion des *répéteurs*. Lorsque l'agent quitte un site, il laisse derrière lui un objet *répéteur* qui est chargé de faire suivre les messages qui lui sont destinés vers son nouveau site. Il offre ainsi une forme de transparence de localisation.

Groupes d'objets

Nous venons de voir que le modèle de communication défini par la librairie *ProActive* améliore considérablement le modèle RMI en lui ajoutant la notion de communication asynchrone entre deux objets distants. Mais le développement d'application distribuée demande plus, comme le fait de pouvoir communiquer avec un nombre important d'objets en une seule fois. Bien entendu, ce type d'opération doit être plus efficace que la simple réplication d'une communication point-à-point entre deux objets. Dans ce but, la librairie *ProActive* offre la possibilité de communiquer avec des groupes d'objets, actifs ou non [BBC07]. Ce mécanisme permet ainsi au développeur de mettre en place facilement une application parallèle suivant le modèle *SPMD*. L'utilisation d'un groupe d'objets actifs d'un même type, appelé *groupe typé* (*typed group*), se fait de la même façon que l'utilisation d'un seul objet de ce type.

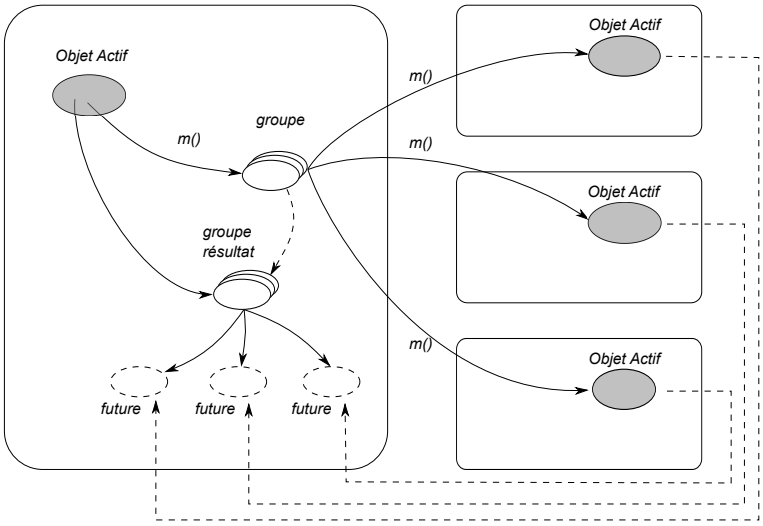


FIGURE 1.2 – Appel d'une méthode sur un groupe d'objets dans une application *ProActive*.

L'appel d'une méthode sur un groupe se transforme en un appel de méthode sur chaque membre du groupe. Si un membre du groupe est un objet actif alors l'appel est réalisé de façon asynchrone et si c'est un objet standard alors il s'agit d'un appel de méthode classique (synchrone). Le

résultat d'un appel de méthode sur un groupe peut être un groupe. le groupe résultat est créé dynamiquement au moment de l'invocation et contient un objet *future* pour chaque membre du groupe sur lequel est appelée la méthode (cf. figure 1.2). C'est une forme de transparence de concurrence.

Le comportement par défaut lors de l'appel d'une méthode sur un groupe est de diffuser (*broadcast*) les paramètres à tous les membres du groupe. Dans certains cas, tous les paramètres ne sont pas utiles à tous les membres mais sont spécifiques à certains membres, le plus souvent en fonction de leur rang. *ProActive* permet de disperser (*scatter*) les paramètres d'un appel de méthode sur un groupe aux différents membres de ce groupe. En d'autres termes, il est possible de transmettre des paramètres différents à des membres d'un même groupe.

1.1.3.4 Le modèle *MapReduce*

Le modèle de programmation *MapReduce* est issu des combinateurs **map** et **reduce** que l'on trouve dans les langages fonctionnels comme *Lisp*. Dans ce type de langage, **map** parcourt une liste d'éléments et applique indépendamment une opération sur chaque élément. *reduce* combine les éléments d'une liste à l'aide d'un opérateur binaire. Pour utiliser le modèle *MapReduce*, le développeur doit définir une fonction *Map* qui traite des données d'entrée et une fonction *Reduce* qui traite les résultats des tâches exécutant la fonction *Map* pour produire le résultat final. La figure 1.3 présente le flux des données lors de l'application du modèle *MapReduce*. Dans un premier temps, les données d'entrée sont divisées en blocs et réparties sur les nœuds de calcul. Les nœuds sont classés en deux catégories : ceux qui exécutent la fonction *Map* (dits *Mapper*) et ceux qui exécutent la fonction *Reduce* (dits *Reducer*). Les nœuds *Mapper* appliquent la fonction *Map* à chaque bloc de données. Le résultat de cette exécution est une liste de paires qui associent à une clé k une valeur v ($list(k, v)$). Ces nouvelles données générées sont appelées les *résultats intermédiaires*. Ensuite, chaque nœud *Reducer* récupère toutes les valeurs v qui sont associées à une clé k et applique la fonction *Reduce* à toutes les valeurs ($(k, list(v))$. Pour finir, les résultats calculés par les nœuds *Reducer* sont assemblés pour donner un résultat final.

Le projet *Hadoop*

Le projet *Hadoop* [Whi10] est un projet *Open Source* distribué par la fondation *Apache*. Ce framework, écrit en *Java*, est adapté au stockage et au traitement par lots de très grandes quantités de données. Il définit son propre système de fichiers, le *Hadoop Distributed File System* (*HDFS*), qui permet de distribuer le stockage de données et d'en faire des analyses grâce au modèle *MapReduce*. Le *HDFS* repose sur deux types de composants majeurs, le *namenode* (le maître) et les *datanodes* (les nœuds de données) :

– Le *namenode* est la véritable pierre angulaire du système de fichier *HDFS*. Il gère l'espace de nommage et l'arborescence du système de fichiers, les métadonnées (noms, permissions, etc...) des fichiers et répertoires. Il centralise la localisation des blocs de données répartis sur

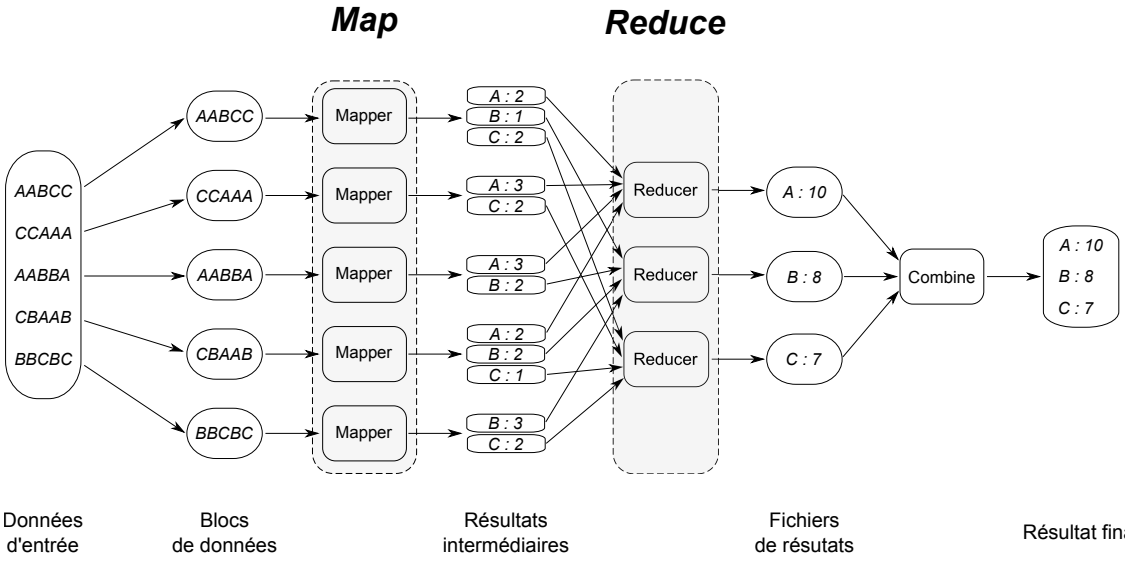


FIGURE 1.3 – Le modèle *MapReduce*. Application du modèle afin de calculer le nombre d'occurrences des lettres dans un texte.

le système. Si le *namenode* disparaît, tous les fichiers sont considérés comme perdus car il n'y aurait alors aucun moyen de reconstituer les fichiers à partir des blocs. Il existe une instance de *namenode* par cluster HDFS. L'historique des modifications dans le système de fichier est géré par une instance secondaire (le *secondary namenode*) cohabitant en backup.

– Les *datanodes* stockent et restituent les blocs de données. Ils communiquent périodiquement au *namenode* la liste des blocs qu'ils hébergent. L'écriture d'un bloc sur un *datanode* peut être propagée en cascade par copie sur d'autres *datanodes*. Le processus de lecture d'un fichier sur le système *HDFS* commence par l'interrogation du *namenode* afin de localiser les blocs sous-jacents. Pour chaque bloc, le *namenode* renvoie l'adresse du *datanode* le plus proche possédant une copie du bloc. L'unité de distance n'est autre que la bande passante disponible ce qui fait que plus la bande passante est importante entre un client et un *datanode*, plus ce dernier est considéré comme proche.

Hadoop définit deux autres types de composants : un *jobtracker* et plusieurs *tasktrackers*. Ils permettent de contrôler le processus d'exécution d'une opération de *MapReduce*, qui est appelée *job* dans *Hadoop*.

– Le *jobtracker* coordonne l'exécution des jobs sur l'ensemble du cluster. Il communique avec les *tasktrackers* en leur attribuant des tâches d'exécution (*Map* ou *Reduce*). Il permet d'avoir une vision globale sur la progression ou l'état du traitement distribué via une console d'administration. Le *jobtracker* est un processus qui s'exécute sur le même nœud que le *namenode*. Il n'y a donc qu'une instance par cluster.
– Les *tasktrackers* exécutent les tâches (*Map* ou *Reduce*) au sein d'une nouvelle JVM qu'ils instancient pour chaque tâche. Ainsi, un crash de la machine virtuelle n'impactera pas le *tasktracker*. Par ailleurs, les *tasktrackers* informent périodiquement le *jobtracker* du niveau de

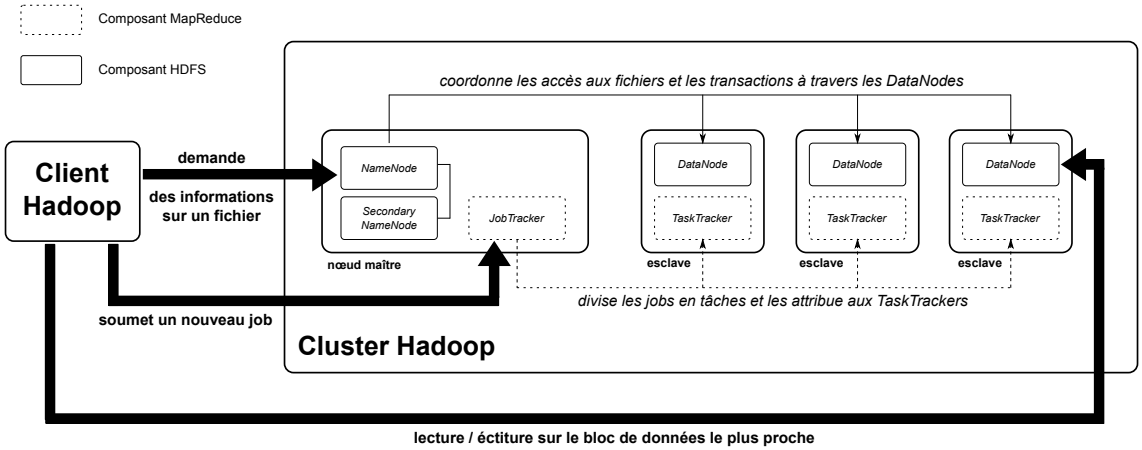


FIGURE 1.4 – Architecture logicielle d'un cluster *Hadoop*.

progression d'une tâche ou bien des erreurs afin que celui-ci puisse reprogrammer et assigner une nouvelle tâche. Un *tasktracker* est un processus qui s'exécute sur la même nœud qu'un *datanode*. Il y a donc autant d'instances que de *datanodes*.

Il est à noter que les communications entre les différents nœuds (*namenode/datanode*, *jobtracker/tasktracker*) s'effectuent par RPC.

1.2 Auto-adaptation de composants logiciels

La présentation des systèmes distribuées de la section 1.1 a mis en avant l'importance des transparences dans la mise en place de tels systèmes. En effet, l'utilisateur ne doit pas être dépendant des aspects techniques et organisationnels qui permettent de réaliser des calculs sur un nombre important de machines. Nous abordons dans cette section l'informatique « autonomique » et nous expliquons dans quelles mesures elle apporte des solutions en matière de transparence à un système distribué.

1.2.1 L'informatique « autonomique »

L'informatique « autonomique » est un axe essentiel de la recherche dans la conception de systèmes informatiques robustes. L'approche proposée par IBM pour définir un système dit « autonomique » [KC03], impose à celui-ci de respecter certaines propriétés, les *auto-** (*auto-configuration*, *auto-réparation*, *auto-optimisation* et *auto-protection*). Le tableau 1.1 liste ces propriétés en présentant les transparences qu'elles apportent dans les systèmes distribués (cf. section 1.1.1). Le respect de ces quatre propriétés assure au système de pouvoir maintenir les services qu'il propose.

Dans un système dit « classique », un administrateur se charge d'effectuer les actions nécessaires pour maintenir les services proposés. Un système autonomique est, quant à lui, doté d'une boucle de contrôle qui automatise ces actions réalisées auparavant par l'administrateur. Le rôle de ce

Propriété auto-*	**Transparences apportées**	**Description**
auto-configuration	*transparence d'échelle*	Les services proposés par le système restent disponibles quelque soit l'environnement dans lequel il évolue.
auto-réparation	*transparence de panne*	Les différentes défaillances matérielles ou logicielles sont analysées par le système qui réalise les opérations nécessaires pour faire en sorte que l'impact soit minime pour les services qu'il propose.
auto-optimisation	*transparence de performance*	Le système surveille les ressources disponibles afin de toujours proposer les meilleurs services, que ce soit au niveau des performances ou des coûts.
auto-protection	*transparence d'accès*	Le système anticipe, détecte, identifie et se protège des attaques contre lui [DPHBB12].

TABLE 1.1 – Tableau récapitulatif des propriétés *auto-** et les transparences qu'elles apportent dans les systèmes distribués.

dernier se réduit à superviser cette boucle de contrôle. Dans [IBM06], IBM définit une boucle de contrôle (cf. figure 1.5) comme étant constituée d'un ensemble de capteurs qui permettent de *surveiller* l'évolution du système (levées d'exceptions, introduction de nouveaux composants, utilisation des ressources, ...). Elle *collecte* les informations auprès de ses capteurs, les *analyse* et définit un plan d'actions à réaliser en fonctions des données collectées. Enfin, les actions sont *exécutées* via des *actionneurs* qui agissent sur le système.

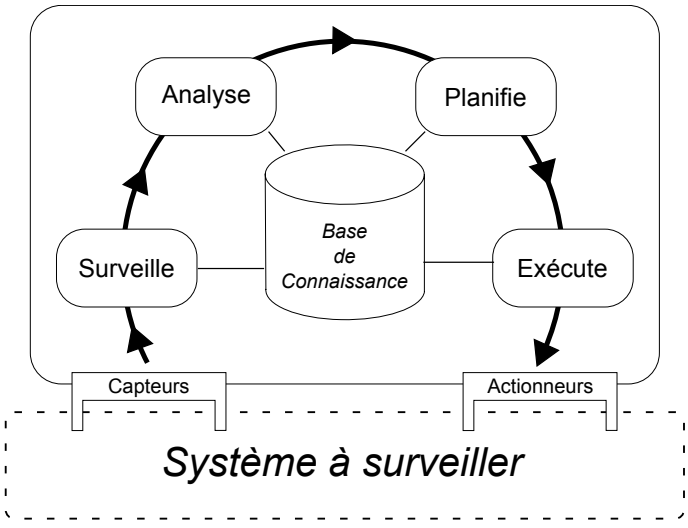


FIGURE 1.5 – Boucle de contrôle dans un système autonomique.

Dans un système autonomique [PH06], une boucle de contrôle peut s'appliquer à un *niveau global*. Dans ce cas, il existe une unique boucle de contrôle ayant une connaissance globale du système. Cette approche facilite l'administration de la boucle de contrôle mais présente un manque de capacité à répondre à une augmentation de la complexité du système. Pour rendre le système plus

flexible, il est possible d'appliquer la boucle de contrôle à un *niveau local*, où chaque composant du système dispose de sa propre boucle de contrôle. Cette approche a l'avantage de supporter l'arrivée de nouveaux composants dans le système mais demeure complexe à mettre en place car toutes les boucles de contrôle doivent alors communiquer entre-elles pour maintenir le système dans un état cohérent.

Dans le domaine qui nous intéresse, celui du calcul distribué, le système est exécuté sur une grille de calcul. D'un coté, le système global doit alors faire face à l'évolution des ressources de calcul qui composent la grille et, de l'autre coté, les ressources de calcul doivent s'adapter aux codes que le système leur demande d'exécuter.

1.2.2 Auto-adaptation aux ressources de calcul

Nombreux sont les outils où l'installation relève du domaine de l'expertise, pire sont ceux où seul l'expert informatique est apte à les employer au quotidien. La démarche d'installation des outils tend à se simplifier par l'emploi de serveur d'applications où les dépendances sont résolues au déploiement. C'est la direction suivie par les bus *ESB* (*Enterprise Service Bus*) où un serveur respectant la norme *OSGi* (*Open Specification Gateway interface*) constitue le cœur de l'application. En revanche, la démarche d'utilisation des outils de simulation reste trop souvent délicate, et peu sujette à l'approximatif. Ainsi, le nombre de ressources de calcul utilisées est une constante dans la configuration de nombreux outils. *SEMC3D* est un outil de simulation électromagnétique dont plusieurs versions existent en Fortran comme *HPF* [LD98]. Chaque cas de calcul nécessite l'écriture d'une description des données mais aussi de l'architecture physique du système distribué sous-jacent. Ces travaux étant des préliminaires au lancement d'un calcul, le coût de chaque exécution est évident. Cela montre l'avantage à autoriser le système à déterminer automatiquement la meilleure configuration d'un cas de calcul et à modifier les paramètres de configuration sans intervention de l'administrateur.

La déclaration de cette configuration n'assure pas de sa validité tout au long de la simulation. Lorsque celle-ci aura débuté, sa validité peut être remise en cause en fonction du contexte d'exécution. La libération de ressource de calcul par une simulation voisine doit permettre d'acquérir de nouvelles ressources de calcul. De même, il peut être nécessaire de diminuer une exploitation de processeurs suite à une demande en provenance d'une autre simulation. Dans ces cas, la configuration doit être revue pour s'adapter aux besoins. Cette adaptation apparaît comme contraire à la recherche de performance, mais elle accroît les possibilités de terminer une simulation ayant pu débuter. Ainsi la plate-forme MASSIM, pour le pilotage de simulations numériques de plasmas, permet d'assembler des simulations. Aussi, les codes manipulés sont naturellement très complexes et nécessitent d'exploiter au mieux les ressources de calcul [Esn05]. Or si les simulations accouplées s'exécutent dans des temps très différents les ressources libérées par l'une ne peuvent être utilisées par l'autre. Nous observons ici un manque évident d'une transparence d'échelle car l'intervention de l'utilisateur est indispensable. Ainsi, l'adaptation d'une simulation

tout au long de son déroulement est une aptitude clé. Nous avons choisi de la placer en tête des besoins utilisateur. Sa prise en compte a un impact direct sur l'aspect tolérance aux pannes.

1.2.3 Auto-adaptation aux codes utilisés

Dans une approche à base de composants, une application est construite comme une organisation de composants logiciels mis en relation par des connecteurs. La notion d'assemblage des composants est définie soit via des interfaces de même type, soit via des liens pour l'adaptation de la connexion des composants. Cette approche nous procure des atouts à toutes les étapes du cycle de vie de la plate-forme de simulation. Tout d'abord lors de son analyse / conception, ce découpage offre une approche incrémentale. Elle permet la construction de la plate-forme par morceau de logiciel. Deuxièmement, lors de la phase de déploiement, ce découpage logiciel permet une répartition de la plate-forme sur plusieurs machines ayant les aptitudes (services) pour les accueillir. De plus, lorsque l'assemblage est fait par échanges de messages, il est alors possible de mettre en place des opérations de gestion logicielle de la plate-forme de simulation par rapport aux mises à jour, correctifs, installation de nouveaux composants, etc.

Enfin, pendant la phase d'exploitation de la plate-forme de simulation, l'approche à base de composants autorise la cohabitation de composants (pour des versions différentes) ou le chargement de nouveaux composants (code de calcul). Aussi, il n'est pas nécessaire que tous les codes numériques soient déjà identifiés lors du lancement d'une simulation. De plus, l'emploi de machines virtuelles ajoute un confort d'exploitation des composants en interprétant des composants écrits dans des langages différents. Ainsi, un composant écrit dans un langage L_1 sera interprété par une machine virtuelle VM_1 sur une machine physique donnée. Ses échanges se feront par messages avec un autre composant écrit dans un langage L_2 interprété dans une machine virtuelle VM_2 locale ou non avec la première.

Ces choix ont déjà été partiellement mis en place dans des logiciels de simulation. SALOME (cf. Section 1.1.3.2), par exemple, est basée sur une architecture ouverte qui respecte les besoins que nous avons énoncés précédemment. Elle constitue aussi le socle logiciel d'autres composants tels que la production de modèles de CAO ou la préparation de données pour des simulations particulières. Elle peut aussi être utilisée comme une plate-forme d'intégration de codes de calcul numérique afin de créer une nouvelle application constituée des composants de base de SALOME et des codes intégrés dans cette plate-forme. Bien entendu, une telle plate-forme repose sur un bus logiciel assurant l'interopérabilité des échanges, celui-ci respecte la norme CORBA, plus délicate et technique que des bus basés sur un service de messagerie XML. Dans le cas d'échanges de données, des aspects de sécurité sont à lui adjoindre. Cet aspect « sécurité des données » peut d'ailleurs avoir un large spectre lorsque les propriétés à assurer sont composites.

1.3 Tolérance aux pannes

Parmi les différentes transparences que nous avons présentées dans la section 1.1.1, la *transparence de panne* fait partie de celles qui joue un rôle clé dans une application pour le calcul distribué. C'est pourquoi nous introduisons dans cette section la notion de tolérance aux pannes ainsi que le vocabulaire relatif à cette notion. Ensuite, nous nous focalisons sur le cas des systèmes distribués en présentant les aspects spécifiques à ce contexte, ainsi que les solutions possibles.

Laprie [ALRL04] définit la sûreté de fonctionnement comme la capacité de fournir un service dans lequel un utilisateur peut raisonnablement placer sa confiance. De façon plus quantitative, la sûreté de fonctionnement permet de décider si un système est capable d'assurer que la fréquence de défaillance du service et la gravité de ces défaillances restent inférieures à un minimum considéré comme acceptable.

Nous avons vu qu'un système distribué peut être considéré comme un ensemble de processus communicants entre eux par messages. On peut distinguer deux types de système distribué, selon les hypothèses faites sur le mode de communication : les systèmes asynchrones, pour lesquels le délai entre l'envoi et la réception d'un message n'est pas borné, et les systèmes synchrones, pour lesquels ce délai est borné et connu.

1.3.1 Classification des pannes

Avant de présenter les différentes méthodes qui rendent un système distribué tolérant aux pannes, il convient de comprendre ce qu'est exactement une panne et comment elle peut être détectée durant l'exécution.

Les pannes qui peuvent survenir durant une exécution distribuée sont classées en quatre catégories :

- les pannes franches (crash, *fail-stop*), que l'on appelle aussi arrêt sur défaillance. C'est le cas le plus simple : on considère qu'un processus peut être dans deux états, soit il fonctionne et donne le résultat correct, soit il ne fait plus rien. Dans le second cas, le processus est considéré comme définitivement défaillant.
- les pannes par omission (transient, omission failures). Dans ce cas, on considère que le système peut perdre des messages. Ce modèle peut servir à représenter des défaillances du réseau plutôt que des processus.
- les pannes de temporisation (timing, performance failures). Ce sont les comportements anormaux par rapport à un temps, comme par exemple l'expiration d'un délai de garde.
- les pannes arbitraires, ou byzantines (malicious, byzantine failures). Cette classe représente toutes les autres pannes : le processus peut alors faire « n'importe quoi », y compris avoir un comportement malveillant.

Cette classification permet de définir des modèles de fautes comme :

– le *modèle de fautes par arrêt* qui implique que le composant qui tombe en panne, se mette dans un état qui permet aux autres composants du système de détecter la panne avant de s'arrêter ;
– ou le *modèle de fautes byzantines* qui implique le composant qui tombe en panne, peut se comporter de manière arbitraire et communiquer aux autres composants du système des informations conflictuelles.

Le cas le plus simple est bien sûr le cas des pannes franches, et on essaie toujours de s'y ramener, par exemple en tuant un processus en cas de comportement imprévu. La plupart des protocoles de tolérance aux pannes pour les systèmes répartis ne considèrent que ce type de pannes, et c'est le cas que nous considérons par la suite.

Le problème est alors de détecter les pannes. Celui-ci est résolu différemment selon le modèle de communication du système : *synchrone* ou *asynchrone*.

Considérons d'abord le cas des modèles synchrones, dans lesquels le temps de transmission d'un message est borné. Supposons qu'un envoi de message provoque la mise en attente de l'émetteur d'une confirmation de réception de la part du récepteur. La détection des pannes franches et de temporisation peut alors être réalisée à l'aide de délais de garde lors des communications : lorsqu'un processus communique avec un autre et ne reçoit pas de confirmation après ce délai de garde, il peut considérer que le processus cible est défaillant.

Dans le cas des modèles asynchrones, la détection d'une panne est plus complexe et porte en partie sur la nature des messages échangés. Ainsi, lorsqu'une requête d'un composant C_1 est traitée par un composant C_2, entrainant une panne de C_2, alors le composant C_1 peut amener d'autres anomalies par manque de réponse. Aussi, la détection de panne peut être réalisée par la mise en place de pattern de messagerie. Un message envoyé peut indiquer qu'il nécessite d'autres messages ou une réponse à un composant identifié. Dans ce cas, la détection de pannes du composant C_2 s'effectue en équipant ce composant d'un gestionnaire de messages. Ainsi, tout message entrainant une réponse pourra au moins recevoir un message d'erreur grâce au respect de l'ordonnancement des messages d'entrée.

1.3.2 Techniques de tolérance aux pannes par réplication

Plusieurs solutions sont proposées pour faire face aux différentes pannes pouvant arriver lors de l'exécution d'une application dans un environnement de type grille de calcul. Cette tolérance aux pannes est toujours réalisée par l'emploi d'un mécanisme de redondance. Les solutions se distinguent alors par le type de redondance qu'elles utilisent : *spatiale* ou *temporelle*.

La *redondance spatiale*, consiste à créer des copies des composants sur divers processeurs. Cette technique permet de masquer les pannes. Pour rendre une grille de calcul complètement fiable, il serait intéressant de dupliquer tous les composants, c'est à dire tous les nœuds de calcul. Mais il est évident que cette solution est très (voire trop) gourmande en ressource et, dans une optique de performance, la duplication de tous les nœuds peut être vue comme du gaspillage. En effet,

l'utilisation de nœuds pour dupliquer des nœuds de calcul diminue le nombre total de nœuds de calcul.

La *redondance temporelle*, appelée aussi *répétition de traitement*, consiste à utiliser du temps de calcul supplémentaire. Cela peut se traduire, par exemple, par la répétition d'une opération autant de fois que nécessaire afin de s'assurer que son exécution soit complète.

L'utilisation d'une réplication spatiale reste adaptée pour les nœuds frontaux et qui ont un rôle central dans l'architecture logicielle utilisée. En effet, cela permet d'assurer la continuité de l'accès aux ressources de calcul. Pour les nœuds de calcul, une réplication temporelle est plus adaptée et suite à une panne, l'application reprend son exécution à partir d'un état antérieur à la panne. La définition de cet état évite à l'application de ré-exécuter tous les calculs depuis son état initial. Nous présentons dans la section suivante cette technique de reprise sur panne.

1.3.3 Techniques de tolérance aux pannes par reprise sur panne

Lors de l'utilisation d'une technique dite de reprise sur panne, l'application distribuée est vue comme un ensemble de processus qui communiquent par échanges de messages. L'état de chacun des processus qui constituent le système distribué forme l'état global de l'application distribuée. Cet état est dit « *cohérent* » si l'application peut atteindre cet état lors d'une exécution correcte, par exemple sans panne. Dans un état global cohérent, pour chaque message échangé entre deux processus, l'état du processus qui envoie le message contient l'envoi du message et l'état du processus réception contient sa réception.

Le principe d'une technique de reprise sur panne est alors de remplacer l'état se trouvant en erreur par un état *cohérent*. Cette technique nécessite la sauvegarde régulière de l'état de l'application. Cet état constitue un point de reprise (*checkpoint*) à partir duquel le système peut repartir et continuer son exécution correctement. La sauvegarde des points de reprise est réalisée à l'aide d'une *mémoire stable*, qui représente un support de stockage « idéal ». Ce stockage doit respecter les conditions suivantes :

– *Accessibilité* : les processus doivent pouvoir accéder à tout moment au support, ceci même en présence de pannes dans l'application.
– *Protection* : les pannes de l'application n'altèrent pas les données écrites sur le support.
– *Atomicité* : les opérations sur le support sont faites de manière atomique.

Ce support est donc utilisé par les processus pour stocker les informations nécessaires pour la reprise d'une application dans un état cohérent suite à une panne.

1.4 Les agents mobiles

Dans le domaine du calcul distribué, la mobilité de code est la capacité pour un programme en cours d'exécution de pouvoir se déplacer, ou migrer, d'une application vers une autre, que

ce soit sur la même machine ou sur un nœud distant. Ce processus qui consiste à déplacer du code à travers les nœuds d'un réseau s'oppose au cas classique d'un calcul distribué où ce sont les données qui sont transférées. Notons enfin que l'utilisation de la mobilité de code dans un système distribué ne doit pas se faire en contrepartie de deux des exigences vues dans la section 1.1.1 que sont les *transparences de mobilité et de localisation.*

Dans cette section, nous définissons plus précisément le paradigme d'agents mobiles, nous examinons ensuite l'infrastructure nécessaire au support de ce type d'agents et enfin nous décrivons la protection à apporter pour rendre sécurisée leur utilisation.

1.4.1 Introduction à la mobilité de code

Une application peut utiliser la mobilité pour exécuter un service. L'exécution d'un service est alors définie par les termes de *composants*, de *sites* et d'*interactions*. Un service est défini par trois composants élémentaires :
- le *savoir-faire* (*code components*) qui correspond au code à exécuter ;
- les *ressources nécessaires* (*resource components*) qui sont utilisées par le service ;
- l'*unité d'exécution* (*computational components*) qui est capable d'effectuer le calcul grâce au code.

Les *interactions* sont les évènements qui impliquent, au moins, deux de ces composants comme par exemple un échange de message. Les *sites* accueillent les composants et sont capables de supporter l'exécution d'une unité d'exécution. Un site représente généralement un nœud sur le réseau. Enfin, le service peut être exécuté uniquement si le savoir-faire, les ressources et l'unité d'exécution sont situés sur le même site.

Deux types de migration sont possibles pour un agent mobile : une migration *réactive*, c'est à dire que l'agent se déplace suite à la demande d'un autre agent, ou une migration *proactive*, dans ce cas l'agent mobile décide lui-même quand et où il doit se déplacer.

Nous pouvons identifier trois principaux paradigmes d'architecture [FPV98] qui définissent les différentes interactions, la coordination et la migration des composants d'un service afin que ce dernier puisse s'exécuter. Il s'agit de l'*évaluation distante*, du *code à la demande* et des *agents mobiles*. Ces trois paradigmes se caractérisent par la localisation des différents composants du service avant et après l'exécution de ce dernier.

Considérons une unité d'exécution A, situé sur un site S_A, ayant besoin du résultat d'un service. Supposons l'existence d'un autre site S_B impliqué dans l'exécution de ce service. Lors d'une *évaluation distante* (cf. figure 1.6), A détient le savoir-faire nécessaire pour exécuter le service mais ne possède pas les ressources nécessaires. Ces dernières se situent sur le site S_B. Par conséquence, A envoie son savoir-faire sur S_B où se trouve une unité d'exécution B, celle-ci exécute le code en utilisant ses ressources disponibles localement et retourne le résultat à A.

Le *code à la demande* (cf. figure 1.7) implique que le composant A dispose des ressources nécessaires sur le site S_A mais ne possède pas le savoir-faire pour les manipuler. A interagit avec

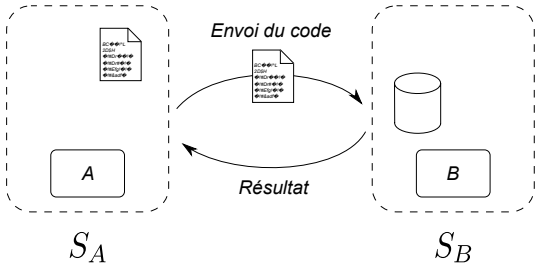


FIGURE 1.6 – L'évaluation distante

B situé sur S_B pour demander le savoir-faire du service. B lui fournit alors ce savoir-faire et A peut ainsi exécuter le service sur le site S_A.

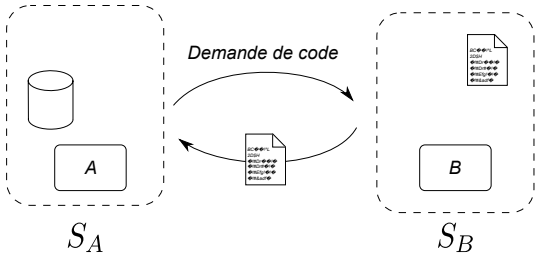


FIGURE 1.7 – Le code à la demande

Enfin, lors de l'utilisation d'une *migration de processus*, le savoir-faire du service se situe sur S_A comme le composant A mais certaines ressources nécessaires se trouvent sur S_B. Au cours de son exécution, A migre vers S_B avec son savoir-faire et ses résultats intermédiaires pour continuer son exécution sur S_B afin d'accéder aux ressources locales.

Si l'évaluation distante et le code à la demande se focalisent uniquement sur le transfert du code entre deux sites, la migration de processus définit une migration totale de l'unité d'exécution (et du code) vers un site distant. On parle alors de *migration forte*.

1.4.2 Le paradigme d'agents mobiles

Le paradigme d'agents mobiles se situe au croisement de deux domaines de recherche : les systèmes multi-agents et la mobilité de code avec plus particulièrement la migration de processus. Le domaine des systèmes multi-agents propose la notion d'agent, Ferber [Fer95] donne la définition suivante :

> « Un agent est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement, qui, dans un univers multi-agents, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents. »

Les agents mobiles apportent à ces agents dits « stationnaires » la capacité de se déplacer sur les différents nœuds d'un réseau [BI02]. Une fois qu'il a migré sur un site distant, un agent mobile

garde alors les mêmes caractéristiques que les agents définis dans les systèmes multi-agents à la différence près qu'il peut une nouvelle fois migrer vers un autre site distant [BR05].

1.4.2.1 Autonomie des agents mobiles

Introduits initialement en 1994 avec l'environnement *Telescript* [Whi99], les agents mobiles sont alors des processus capables de se déplacer, de leur propre initiative, sur les différents nœuds d'un réseau afin de travailler localement sur les ressources. Un agent mobile est un agent capable de migrer de manière autonome sur les différents sites d'un réseau. Cette migration est dite « *proactive* » car le déplacement de l'agent est à l'initiative même de l'agent mobile. Ce type de migration permet de garder à l'agent son caractère autonome. A l'inverse, lors d'une migration « *réactive* », c'est le système qui initie les déplacements de l'agent mobile. L'autonomie des agents est alors perdue mais ce type de migration permet de rendre transparents les déplacements de l'agent. Nous pouvons noter que l'utilisation de ce type de migration dans un environnement homogène ne nuit pas réellement à l'autonomie de l'agent, en effet après son déplacement l'agent se retrouve dans le même environnement

1.4.2.2 Apports des agents mobiles

Pour répondre aux exigences des systèmes distribués dans un environnement tel que les grilles de calcul, les agents mobiles sont particulièrement adaptés pour deux raisons : la *recherche de performance* et la *facilité d'adaptabilité*.

Recherche de performance

La migration du code vers les données offerte par l'utilisation d'un agent mobile :
– réduit la latence de certaines étapes liée à l'utilisation réseau,
– évite les transferts de données intermédiaires à travers le réseau lors d'un calcul,
– permet de continuer l'exécution du calcul malgré la présence de coupures du réseau.
Si ces différents avantages permettent le plus souvent à un agent mobile de réaliser des calculs plus rapidement qu'avec une solution traditionnelle de type client/serveur, il arrive que l'utilisation d'un agent mobile puisse ralentir l'exécution d'un calcul. En effet, le code de l'agent peut être plus important (en nombre de bytes) que les données avec lesquelles il travaille. Dans ce cas le transfert de l'agent est plus long que le transfert des données. De la même façon, si le réseau offre des temps de transfert rapides alors l'exécution d'un agent mobile peut être plus lente que le transfert des données. Ceci est dû au fait que les agents mobiles sont souvent implantés à l'aide de langages interprétés pour des raisons de portabilité et de sécurité, l'interprétation du code peut alors être plus lente que le transfert de données. L'apport des agents mobiles dans un système distribué est donc important si le réseau n'est pas rapide, ce qui est souvent le cas dans un environnement comme une grille de calcul.

De plus, comme nous l'avons vu dans la section 1.1, une grille de calcul est composée de machines hétérogènes. Dans un tel environnement, l'utilisation d'agents mobiles offre un avantage considérable. En effet, lors de son exécution, il se déplace vers les machines disponibles les plus performantes pour profiter du maximum de puissance de calcul offert par le grille.

Facilité d'adaptation

Une application construite à base d'agents mobiles peut se re-configurer dynamiquement en réaction à une situation particulière pour améliorer la performance, satisfaire la tolérance aux pannes ou réduire le trafic réseau. En effet, la capacité d'adaptation individuelle des agents mobiles permet de faire évoluer un système distribué pour répondre aux besoins d'adaptation qu'ils soient opérationnels (décentralisation des données, hétérogénéité matérielle et logicielle des composants ...) ou fonctionnels (spécialisation des calculs à réaliser au niveau des données ou avec les autres agents). Bien entendu, l'adaptation d'un agent doit s'effectuer de manière transparente du point de vue de l'application, et donc de l'utilisateur. Les apports majeurs de ce type d'architecture concernent donc la facilité de mise en œuvre de l'adaptation dynamique aux conditions d'exécution et au calcul à effectuer.

1.4.3 Sécurité dans les systèmes d'agents mobiles

L'accueil d'un agent mobile par un processus hôte est une action qui comporte un risque maximal si la sécurité n'est pas prise en compte. De nombreux travaux de recherche relatifs à la sécurité des agents mobiles ont été réalisés. Ils se divisent sur deux axes [BCF+05]. Le premier concerne la protection de l'hôte contre des agents mobiles malveillants tandis que le second se concentre sur la protection de l'agent mobile contre la malveillance de l'hôte sur lequel il s'exécute.

1.4.3.1 Protection de l'hôte accueillant un agent mobile

La protection de l'hôte est sujet à un certain nombre de mesures protégeant le système hôte de l'agent mobile. Parmi les techniques les plus connues, nous pouvons citer la signature du code mobile [RV00], le bac à sable (*sandboxing*), le code avec preuve (*proof carrying code*) [NL98], l'estimation de l'état (*state appraisal*) [FGS96] et l'historique des hôtes (*path histories*) [CGH+98].

La signature du code - Cette technique permet d'obtenir une authentification de haut niveau pour les hôtes. Elle assure aussi l'intégrité du code pour l'hôte visité. Une signature digitale sert donc de moyen de confirmation de l'authenticité de l'agent mobile, de son origine et de son intégrité. Le signataire du code est soit son créateur ou une autre entité ayant modifiée le code. Un agent autonome opère à la place de l'utilisateur local de l'hôte. Les systèmes d'agents mobiles utilisent la signature comme une indication de l'autorité sous laquelle l'agent mobile s'exécute.

La technique du bac à sable - Cette technique fait référence aux bacs à sable (*sandbox*) utilisés par les démineurs pour faire exploser les engins en toute sécurité. Il s'agit d'exécuter le code de l'agent

mobile dans un environnement restreint (*sandbox*) mais qui apparaît à l'agent mobile comme étant le système dans lequel il s'exécute [GD10]. Elle interdit, par exemple, l'accès au système de fichiers, l'ouverture d'une connexion réseau, l'accès à des propriétés ou à des programmes du système local. Ce contexte est construit par le système de contrôle d'accès aux ressources et dépend de la politique choisie par le système local. Les agents ont des privilèges limités et peuvent être exécutés de manière sécurisée. Ainsi un hôte peut exécuter un agent suspect dans le bac à sable sans trop se soucier des problèmes de sécurité du système local.

Le code avec preuve - Cette technique permet au système hôte de vérifier formellement l'agent mobile à l'aide de propriétés qui accompagnent l'agent lors de sa migration. L'agent mobile doit fournir au système la preuve qu'il est conforme à une certaine politique de sécurité. Dans ce cas, le système hôte vérifie cette preuve et peut exécuter l'agent en toute sécurité. L'inconvénient majeur de cette technique est la difficulté à générer de telles preuves formelles de manière automatisée et efficace.

L'estimation de l'état - Cette technique tente de faire en sorte que l'état de l'agent reste intacte afin de s'assurer que l'agent mobile n'exécute pas d'actions malveillantes sur l'hôte. L'agent mobile migre alors avec une fonction d'évaluation de son état créée par son auteur. L'hôte utilise cette fonction pour vérifier que l'agent se trouve dans un état correct. Cette technique reste peu courante car il est difficile de capturer et de décrire les états d'un agent de façon efficace et sécurisée.

L'historique des hôtes - L'idée est ici de permettre à un hôte de connaître la liste des hôtes sur lesquels l'agent mobile a été exécuté précédemment. Si l'agent mobile a été exécuté précédemment sur un hôte non approuvé, le nouvel hôte peut décider de ne pas l'exécuter ou de restreindre ses privilèges d'exécution. Cette technique nécessite que chaque hôte ajoute une entrée signée dans l'historique pour indiquer son identité.

1.4.3.2 Protection de l'agent mobile

La protection de l'agent mobile contre un hôte malveillant demeure un problème ouvert et difficile dû au fait que l'environnement d'exécution a un contrôle total sur l'agent mobile (sans quoi, la protection de l'hôte ne serait pas possible). Les solutions proposées abordent ces menaces d'une manière totale ou partielle et visent essentiellement à rendre les attaques de l'hôte inutiles ou détectables. Un hôte malveillant peut essayer d'attaquer un agent mobile pour altérer le code qu'il contient, obtenir un service gratuitement ou accéder aux données de l'agent pour y consulter ou manipuler des informations privées. L'agent mobile est vulnérable pendant son exécution sur la plate-forme hôte. Par conséquent, l'agent mobile exige des garanties concernant sa protection contre les menaces des hôtes malveillants. L'agent mobile doit se protéger contre n'importe quel acte visant à sa détérioration, sa destruction ou la manipulation de son code, de son état ou de ses données.

Un agent mobile est exposé à diverses menaces lorsqu'il se trouve sur un hôte sur lequel il vient de se déplacer [BC02]. Ce problème est difficile car l'environnement visité a un contrôle total sur l'exécution de l'agent mobile. Pour comprendre ce que risque un agent mobile lors de son exécution sur un site malveillant, nous pouvons rappeler les éléments qui constituent un agent mobile et qui peuvent être des cibles d'attaque pour un hôte malveillant [LMR00] :

- le *savoir-faire* : ensemble des instructions permettant l'exécution de l'agent,
- les *ressources nécessaires* :
  - *les données statiques* : données non modifiées durant l'exécution de l'agent,
  - *les données collectées* : ensemble des résultats obtenus au cours de l'exécution (sur les différentes localités explorées) de l'agent,
  - *l'état courant* : ensemble de données servant à l'exécution courante de l'agent.

Un agent mobile ne souhaite pas que son hôte puisse avoir accès à des informations critiques qu'il contient. Par exemple, un hôte malveillant pourrait récupérer la signature d'un code et l'utiliser pour créer un nouvel agent afin de s'introduire dans des environnements auxquels il n'a normalement pas accès.

Pour le code, un agent transporte un savoir-faire propre à son concepteur qui pourrait tomber aux mains de ses concurrents. Il est possible de distinguer trois grandes catégories d'attaques que des hôtes malveillants pourraient mener : l'inspection (espionnage de données et du code), la modification (altération des données et du code) et le rejeu (la réexécution du code) [Zac03]. L'inspection consiste à examiner le contenu de l'agent, ou le flot d'exécution afin de récupérer des informations sensibles transportées par l'agent mobile. La modification est réalisée en remplaçant certains éléments (donnée ou partie de code) de l'agent mobile dans le but de conduire une attaque. Le remplacement du code incitera l'agent à effectuer des opérations malveillantes sur les futurs hôtes à visiter. Tandis que le remplacement des données permet à l'hôte malicieux de manipuler l'agent mobile à son avantage. Le rejeu, quant à lui, s'obtient en clonant l'agent puis en exécutant le clone dans plusieurs configurations pour retrouver le savoir de l'agent.

La protection d'un agent mobile à l'encontre d'hôtes malveillants revient à protéger principalement son exécution, son intégrité et sa confidentialité. Pour protéger son exécution, il faut s'assurer que son hôte ne puisse pas le détourner vers d'autres destinations non voulues, le priver de ressources ou faire en sorte que son exécution se termine prématurément. Assurer l'intégrité d'un agent mobile implique de détecter toute modification de son code et de son état qui pourrait être due à une mauvaise exécution de son hôte [BMW98], [MB02]. Enfin, pour garder la confidentialité, il est nécessaire de cacher le code et l'état de l'agent mobile vis-à-vis de l'hôte qui l'exécute [ST98], [ACCK01], [LAFH04].

1.5 Bilan des besoins

Nous avons présenté dans ce chapitre le contexte de recherche dans lequel se situe cette thèse. L'utilisation de grille de calcul donne à ses utilisateurs une grande puissance de calcul mais nécessite de mettre en place des applications distribuées capables de profiter des ressources disponibles. Différents modèles de programmation existent pour définir l'architecture d'une application exécutée dans un environnement distribué et les besoins qui en ressortent peuvent se classer en terme de *transparences* dont les plus recherchées sont :

- La *transparence d'échelle* : l'environnement d'exécution d'une grille de calcul peut évoluer durant un cas de calcul ;
- La *transparence de panne* : la durée d'un cas de calcul est telle que les ressources ne sont pas suffisamment fiables pour être disponibles durant tout le calcul ;
- La *transparence de performance* : la composition d'une grille de calcul peut être très hétérogène, il est donc important de faire en sorte que les ressources les plus performantes soient utilisées.

La recherche des transparences les plus adaptées à l'utilisateur et au cas de calcul est un thème auquel nous sommes attachés. Obtenir l'ensemble des transparences que nous avons évoquées est encore un idéal, mais définir le meilleur compromis et le mettre en œuvre est notre objectif. Nos expériences passées, conjuguées aux travaux de recherche mentionnés dans ce chapitre, nous donnent matière à structurer les exigences des cas de calculs que nous traitons. Les transparences d'accès et de localisation offrent les aptitudes pour gérer de manière simple une grille de calcul. La transparence de mobilité apporte l'adaptabilité qui manque le plus souvent aux plates-formes existantes. Dans ce but nous nous orientons vers une architecture utilisant des agents mobiles. Ce choix apporte une solution à la transparence de performance en exploitant au mieux les ressources de calcul disponibles dans une grille de calcul.

Nous définissons dans le chapitre suivant une architecture logicielle répondant à ces exigences en utilisant un algèbre de processus, nommé π-calcul.

Chapitre 2

Modélisation formelle d'une architecture logicielle pour la simulation numérique

Sommaire

Une algèbre de processus est un formalisme mathématique qui permet l'étude des systèmes concurrents. De nombreux auteurs, comme Robin Milner ont proposé des algèbres tels que le π-calcul [MPW92, Mil99] comme un langage formel pour spécifier et analyser des processus mobiles. La quantité et la qualité du travail réalisé dans le domaine des calculs de processus mobiles, nous permettent d'avancer que le π-calcul est devenu un choix reconnu pour décrire formellement le parallélisme, l'interaction et la mobilité.

Dans ce chapitre, nous décrivons π-calcul polyadique d'ordre supérieur puis nous définissons formellement une architecture logicielle d'un système capable de s'adapter à son environnement d'exécution dans le but de résoudre des cas de calcul numérique.

2.1 Le π-calcul

Le π-calcul est une algèbre de processus au même titre que CSP [Hoa86] ou CCS [Mil89]. Ces algèbres proposent des méthodes formelles afin de modéliser des interactions entre processus. Les modélisations réalisées permettent de construire un modèle mathématique afin de garantir la cohérence du modèle et la conformité du programme. Dans le π-calcul, un terme bien formé dénote un processus et les processus se synchronisent et échangent des noms. Notons que le π-calcul ne fait aucune hypothèse sur les vitesses d'exécution relatives des différents processus, qui sont donc présumés progresser chacun à leur rythme.

Dans cette section, nous détaillons différentes versions du π-calcul. Nous commençons par une introduction en présentant sa forme la plus simple, le π-calcul monadique (Section 2.1.1), nous continuons par sa forme polyadique (Section 2.1.1.5). Enfin, nous terminons par le π-calcul d'ordre supérieur (Section 2.1.2) qui nous permet de modéliser des agents mobiles (cf. Section 1.4).

2.1.1 Introduction au π-calcul

Nous commençons cette section par quelques définitions et conventions concernant le π-calcul monadique du premier ordre. Nous finissons par un exemple simple pour introduire la notion de mobilité offerte par le π-calcul.

2.1.1.1 Conventions de nommage

Nous considérons une ensemble infini de noms N, notés $a, b, \ldots, z$, qui représentent des canaux de communications, des données et des variables, et un ensemble de termes (nous les nommons également processus ou agent), noté typiquement P , Q , R ..., qui représentent des processus. A partir du tableau 2.1, nous pouvons noter qu'un agent π-calcul peut prendre les formes suivantes :

- $\alpha.P$ dénote la préfixation d'un processus P par une action α. Le préfixe α peut prendre les formes suivantes :
  - $a(x)$ dénote le réception de la valeur, stockée dans la variable x, via le canal a. Ce préfixe lie le nom x au processus P.
  - $\overline{a}\,\langle\, x \,\rangle$ dénote l'émission de la valeur, stockée dans la variable x, via le canal a. Ce préfixe ne lie pas le nom x au processus P.
  - τ dénote une action interne, non observable.
- $P \mid Q$ met en parallèle deux processus P et Q. Les processus P et Q peuvent agir indépendamment l'un de l'autre mais peuvent aussi agir l'un sur l'autre.
- $P + Q$ permet le choix entre le processus P et le processus Q.
- $!\, P$ peut être considéré comme une composition infinie $P \mid P \mid \cdots \mid P$.
- $[x = y]\, P$ définit que le processus évolue comme P si x et y sont identiques.

Préfixes	α	::=	$\overline{a}\langle x \rangle$	Émission
			$a(x)$	Réception
			τ	Interne
Agents	P	::=	0	Nul
			$\alpha . P$	Préfixation
			$P + P$	Somme
			$P \mid P$	Parallèle
			$[x = y]P$	Égalité
			$[x \neq y]P$	Inégalité
			$(\nu x) P$	Restriction
			$!P$	Réplication
			$A(y_1, \ldots, y_n)$	Identifiant
Définitions			$A(x_1, \ldots, x_n) \stackrel{def}{=} P$	où $i \neq j \Rightarrow x_i \neq x_j$

TABLE 2.1 – La syntaxe du π-calcul.

L'identifiant $A(x_1, \ldots, x_n)$, où n est l'arité de A, permet la définition d'un processus. Chaque identifiant a une définition tel que $A(x_1, \ldots, x_n) \stackrel{def}{=} P$ où les noms x_i sont distincts. Cette définition peut être considérée comme une déclaration d'agent avec $x_1, \ldots, x_n$ en tant que paramètres formels. La notation $A(y_1, \ldots, y_n)$ se comporte comme P où y_i remplace x_i pour chaque i et peut être considérée comme une invocation d'un agent avec les paramètres effectifs $y_1, \ldots, y_n$. Ces définitions peuvent être récursives mais un noyau du π-calcul ne nécessite pas de définitions récursives.

L'opérateur ν permet de restreindre un nom à un processus. $(\nu x) P$ signifie que x n'est connu que du processus P, c'est à dire que la portée du nom x est limitée à P. Toutefois, rien n'empêche la portée d'une restriction d'envoyer le nom à un autre processus. Par exemple, si $P \stackrel{def}{=} (\nu y)\, \overline{x}y \,.\, P'$ et $Q \stackrel{def}{=} x(z) \,.\, Q'$ alors l'évaluation de l'expression $P \mid Q$ signifie que Q reçoit le nom y a travers le canal x et Q' connaît y qui est pourtant restreint à P. Il s'agit d'une *expulsion de portée* (*scope extrusion* en anglais).

Nous avons noté qu'il existe deux opérateurs de liaison qui permettent de lier une variable à un processus : le préfixe de réception $a(x)$ et la restriction (νx) . Pour un processus P, nous pouvons alors définir l'ensemble des noms libres noté $fn(P)$ (*free names*) et l'ensemble des noms liés noté $bn(P)$ (*bound names*) tels que :

Nous pouvons alors définir que les noms d'un processus P sont $n(P) \stackrel{def}{=} bn(P) \cup fn(P)$. Dans la définition $A(x_1, \ldots, x_n) \stackrel{def}{=} P$, nous supposons que $fn(P) \subseteq \{x_1, \ldots, x_n\}$.

$$
\begin{array}{ll}
bn(0) = \emptyset & , fn(0) = \emptyset \\
bn(x(y)\,.\,P) = \{y\} \cup bn(P) & , fn(x(y)\,.\,P) = \{x\} \cup fn(P) \setminus \{y\} \\
bn(\overline{x}\,\langle\, y \,\rangle\,.\,P) = bn(P) & , fn(\overline{x}\,\langle\, x \,\rangle\,.\,P) = \{x, y\} \cup fn(P) \\
bn((\,\nu\, x\,)\ P) = \{x\} \cup bn(P) & , fn((\,\nu\, x\,)\ P) = fn(P) \setminus \{x\} \\
bn(P \mid Q) = bn(P) \cup bn(Q) & , fn(P \mid Q) = fn(P) \cup fn(Q) \\
bn(!P) = bn(P) & , fn(!P) = fn(P)
\end{array}
$$

2.1.1.2 Congruence structurelle

Une *congruence* est une relation intéressante à définir pour les équivalences comportementales. Cette propriété permet par exemple de décomposer les processus étudiés en sous-termes, de prouver l'équivalence sur cette décomposition et d'en déduire l'équivalence des termes initiaux par congruence. La *congruence structurelle*, notée $\equiv$, est la plus petite congruence qui satisfait les règles suivantes :

1. Si P et Q ne diffèrent que par un changement de noms liés alors $P \equiv Q$;
2. Les règles définies par le monoïde commutatif [2] pour $\mid$:

$$P \mid Q \equiv Q \mid P,\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R,\ P \mid 0 \equiv P$$

3. Les règles définies par le monoïde commutatif pour $+$:

$$P + Q \equiv Q + P,\ P + (Q + R) \equiv (P + Q) + R,\ P + 0 \equiv P$$

4. Les règles d'extension de la portée des restrictions suivantes :

$$
\begin{array}{lll}
(\,\nu\, x\,)\ 0 & \equiv 0 & \\
(\,\nu\, x\,)\ (P \mid Q) & \equiv P \mid (\,\nu\, x\,)\ Q & \text{si } x \notin \text{fn}(P) \\
(\,\nu\, x\,)\ (P + Q) & \equiv P + (\,\nu\, x\,)\ Q & \text{si } x \notin \text{fn}(P) \\
(\,\nu\, x\,)\ [u = v]\, P & \equiv [u = v]\, (\,\nu\, x\,)\ P & \text{si } x \neq u \text{ et } x \neq v \\
(\,\nu\, x\,)\ [u \neq v]\, P & \equiv [u \neq v]\, (\,\nu\, x\,)\ P & \text{si } x \neq u \text{ et } x \neq v \\
(\,\nu\, x\,)\ (\,\nu\, y\,)\ P & \equiv (\,\nu\, y\,)\ (\,\nu\, x\,)\ P &
\end{array}
$$

5. $!P \equiv P \mid !P$

2. Soit E un ensemble, $*$ une opération associative et e un élément neutre pour $*$ alors $E(E, *, e)$ est un monoïde. Si la loi $*$ est commutative, le monoïde est dit commutatif.

2.1.1.3 Sémantique opérationnelle

Une sémantique opérationnelle est très utile pour calculer l'évolution d'un terme. La sémantique opérationnelle de π-calcul est donnée par un système de transitions étiquetées, où les transitions sont du type $P \xrightarrow{\alpha} P'$ avec α un ensemble d'actions pouvant être l'action interne τ , l'action de réception $a(x)$ et l'action d'émission $\overline{a}\langle x \rangle$. Cette relation de réduction signifie que P est transformé en P' suite à l'action α.

La transition $a(x).P \xrightarrow{a(u)} P\{u/x\}$ signifie que si le nom u est transmis à travers le canal a alors le processus $a(x).P$ qui attend un nom sur le canal a reçoit le nom u et effectue une substitution [3] de x par u et se comporte ensuite comme P, où toutes les occurrences de x sont remplacées par u.

Les règles de communication sont les plus spécifiques pour un système ayant un besoin de mobilité. L'interaction entre deux processus est donnée par les règles COM_1 et COM_2 de communication :

$$\mathrm{COM}_1 \ : \ \frac{P \xrightarrow{\overline{a}\langle u \rangle} P',\ Q \xrightarrow{a(x)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{u/x\}} \qquad \mathrm{COM}_2 \ : \ \frac{P \xrightarrow{a(x)} P',\ Q \xrightarrow{\overline{a}\langle u \rangle} Q'}{P \mid Q \xrightarrow{\tau} P'\{u/x\} \mid Q'}$$

La notation d'une sémantique opérationnelle est donnée dans le tableau 2.2.

2.1.1.4 Un premier exemple de mobilité

La grande différence entre le π-calcul et les algèbres de processus qui l'ont précédé concerne la *mobilité*. En effet, le π-calcul offre la possibilité de transmettre le nom d'un canal à travers un autre canal. Par exemple, considérons un système composé de trois processus P, Q et R qui s'exécutent en parallèle tel que :

$$System \stackrel{def}{=} P \mid Q \mid R \ \text{ avec } \begin{cases} P \stackrel{def}{=} \overline{a}\langle x \rangle \,.\, P' \\ Q \stackrel{def}{=} x(y) \,.\, Q' \\ R \stackrel{def}{=} a(z) \,.\, (\nu\, v)\, \overline{z}\langle v \rangle \,.\, R' \end{cases} \tag{2.1}$$

alors le système se réduit de la manière suivante :

$$P \mid Q \mid R \xrightarrow{\tau} P' \mid x(y) \,.\, Q' \mid \overline{x}\langle v \rangle \,.\, R' \xrightarrow{\tau} P' \mid Q' \mid R' \tag{2.2}$$

3. Une *substitution* est une fonction de l'ensemble des noms vers d'autres noms. La notation $\{x/y\}$ signifie la substitution du nom y par le nom x.

STRUCT	$\dfrac{P \equiv P', P \xrightarrow{\alpha} Q, Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$
PREFIX	$\dfrac{}{\alpha.P \xrightarrow{\alpha} P}$
SUM	$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$
MATCH	$\dfrac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}$
PAR	$\dfrac{P \xrightarrow{\alpha} P', \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$
COM	$\dfrac{P \xrightarrow{a(x)} P',\ Q \xrightarrow{\overline{a}\langle u \rangle} Q'}{P \mid Q \xrightarrow{\tau} P'\{u/x\} \mid Q'}$
RES	$\dfrac{P \xrightarrow{\alpha} P', x \notin \alpha}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}$

TABLE 2.2 – La sémantique opérationnelle du π-calcul.

Le système transmet le canal x à travers le canal a du processus P au processus R. Il y a alors substitution de toutes les variables avec le nom z par le nom x dans le processus R. Cela permet au processus R de transmettre au processus P la variable avec le nom v à travers le canal x.

2.1.1.5 Le π-calcul polyadique

Le π-calcul polyadique [Mil91] permet la transmission d'un tuple de valeurs lors d'une interaction entre deux processus avec l'action de réception $a(\, x_1 \, \cdots \, x_n \,)$ et l'action d'émission $\overline{a}\,\langle\, y_1 \, \cdots \, y_n \,\rangle$ où les x_i sont distincts. Les notations sont le prolongement naturel de la syntaxe du π-calcul monadique. Par exemple, l'écriture suivante

$$a(\, \vec{x} \,) \;.\; P \mid \overline{a}\,\langle\, \vec{y} \rangle \;.\; Q \xrightarrow{\tau} P\{\vec{y}/\vec{x}\} \mid Q$$

signifie qu'il y a une substitution simultanée de tous les noms y_i par les noms x_i.

2.1.2 Le π-calcul d'ordre supérieur

La transmission d'un canal par le biais d'un autre, possible avec le π-calcul du premier ordre, offre une première approche de la mobilité. Avec le π-calcul d'ordre supérieur [San92], noté HOπ (pour High-Order π-calculus), la mobilité va plus loin que le simple envoi d'un canal. En effet, un processus peut transiter au travers d'un canal afin de fournir un nouveau comportement à un processus en cours d'exécution.

La syntaxe du HOπ est une extension de la syntaxe du π-calcul :

$$P ::= \ \overline{x}\langle K \rangle \ . \ P \mid x(U) \ . \ P \mid P \ \Big| \ Q \mid \tau \ . \ P \mid (\nu \, x) \ P \mid [x = y]P \mid [x \neq y]P \mid !P \mid 0$$

avec K un processus et U une variable pouvant accueillir un processus.

Par exemple, avec le HOπ il est possible d'écrire le système (eq. 2.3) suivant où le processus P se déplace à travers le canal x.

$$\overline{x}\langle P \rangle \ . \ Q \ \Big| \ x(X) \ . \ X \xrightarrow{\tau} Q \ \Big| \ P \tag{2.3}$$

Pour le HOπ, aucune modification des règles de congruence structurelle n'est nécessaire. En ce qui concerne la sémantique opérationnelle, seules les règles de communication sont affectées car les valeurs échangées ne sont plus seulement des noms mais peuvent aussi être des termes d'ordre supérieur :

$$\text{COM\_HO}_1 \ : \ \frac{P \xrightarrow{\overline{x}\langle K \rangle} P', \ Q \xrightarrow{x(U)} Q'}{P \ \Big| \ Q \xrightarrow{\tau} P' \Big| \ Q'\{K/U\}} \qquad \text{COM\_HO}_2 \ : \ \frac{P \xrightarrow{x(U)} P', \ Q \xrightarrow{\overline{x}\langle K \rangle} Q'}{P \ \Big| \ Q \xrightarrow{\tau} P'\{K/U\} \Big| \ Q'}$$

Pour les deux règles de sémantique COM_HO$_1$ et COM_HO$_2$, il est nécessaire de noter que le nombre de paramètres de U est identique à celui de K.

Modélisation d'un agent mobile

L'utilisation du HOπ nous permet alors de modéliser un système contenant un agent mobile comme le présente le système suivant :

$$System \ \stackrel{def}{=} \ (\nu \, x) \ \Big(A(x) \ \Big| \ B(x) \Big) \text{ avec } \begin{cases} C(w) \ \stackrel{def}{=} \ \tau \ . \ C' \ . \ 0 \\ A(z) \ \stackrel{def}{=} \ (\nu \, y) \ \overline{z}\langle C(y) \rangle \ . \ A' \\ B(x) \ \stackrel{def}{=} \ x(Q(u) \ ) \ . \ Q(u) \ . \ B' \end{cases}$$

Ici, le système se réduit de la manière suivante :

$$System \xrightarrow{\tau} Q(u) \,.\, B' \mid A' \xrightarrow{\tau} C' \,.\, B' \mid A' \tag{2.4}$$

Le processus C modélise un agent mobile. Cet agent migre à travers le canal x commun au processus A et B. Le processus A envoie l'agent mobile C au processus B. B continue son exécution comme C et à la fin de l'exécution de C il est équivalent au processus B'.

Nous utilisons, dans la section suivante, l'expressivité fournie par le HOπ pour définir une architecture logicielle pour la calcul parallèle.

2.2 Modélisation d'une architecture logicielle pour le calcul parallèle

Dans cette section, nous définissons une architecture logicielle pour la mise en place d'un espace de travail dédié à la résolution de cas de calcul numérique. Cet espace de travail a pour but d'accueillir des cas de calcul numérique, d'assurer leurs exécutions et de pouvoir en faire l'administration. Ce dernier point est d'ailleurs essentiel car notre objectif n'est pas la recherche de la meilleure performance pour chacun des cas, mais la capacité de pouvoir gérer leur cycle de vie : création, réalisation, suspension, reprise.

Le chapitre 1 a mis en évidence l'importance de l'auto-adaptation d'une architecture logicielle à son environnement d'exécution surtout lorsque celui-ci est une grille de calcul. En effet, ce type d'environnement est constitué d'un nombre variable de ressources lors de l'exécution d'un cas de calcul. La section 1.2 ajoute aussi la nécessité à ces ressources de s'adapter aux codes utilisés sur la grille. Notre architecture logicielle, et donc notre espace de travail, doivent respecter ces propriétés.

Avant de définir formellement cet espace de travail (Section 2.2.4), nous commençons par modéliser un exemple simple, comme celui du modèle *Maître-Travailleurs* (Section 2.2.1). Ensuite, nous présentons la modélisation d'une « ferme de travailleurs » (Section 2.2.2) capable de participer à la résolution d'un cas de calcul, que nous modélisons dans la section 2.2.3.

2.2.1 Modélisation d'une exécution suivant le modèle *Maître-Travailleurs*

Pour commencer la modélisation de notre architecture logicielle, nous partons de la modélisation d'un cas simple, celui du modèle *Maître-Travailleurs* (cf. Section 1.1.3). Dans ce modèle, deux types de processus participent à la résolution d'un cas de calcul : un *processus maître* et des *processus travailleurs*. Le principe est simple : le processus maître divise le problème initial en n tâches et les distribue aux processus travailleurs. Ces derniers traitent alors les tâches qui leur sont attribuées et retournent les résultats au processus maître. Celui-ci peut ainsi produire le résultat final.

2.2.1.1 Définition des processus « travailleurs »

Nous définissons une tâche par le terme $Task$ (eq. 2.5). Le code exécuté par une tâche est lui modélisé par l'action silencieuse τ.

$$Task \stackrel{def}{=} \tau \,.\, 0 \tag{2.5}$$

Une tâche est exécutée par un processus travailleur. Ce processus est modélisé par le terme $Worker$ (eq. 2.6). Ce terme est paramétrable par le canal *execute*.

$$Worker(\, execute\,) \stackrel{def}{=} execute(\, T\,) \,.\, T \tag{2.6}$$

Grâce au pouvoir d'expression du HOπ (cf. section 2.1.2), un agent $Worker$ reçoit à travers son canal *execute* un agent noté T (eq. 2.6). L'agent $Worker$ continue alors son évaluation comme T. Nous pouvons écrire la réception d'une tâche par un agent $Worker$ comme la réduction du terme $Worker$ (eq. 2.7) par l'emploi de la définition de la sémantique opérationnelle définie dans la section 2.1.1.3) avec

$$Worker(\, execute\,) \xrightarrow{execute(\, Task_1\,)} Task_1 \text{ avec } Task_1 \stackrel{def}{=} Task \tag{2.7}$$

L'agent $Task_1$ (eq. 2.7) peut être considéré comme un agent mobile. Nous pouvons d'ailleurs noter ici que ce n'est pas la définition d'un agent qui lui donne sa propriété mobile mais le contexte d'exécution d'un agent.

Les communications à travers le canal *execute* sont des communications point-à-point unidirectionnelles et la section 2.2.1.2 présente le terme $Master$ qui émet sur ce canal.

Soit l'abréviation[4] $Worker_i$ telle que $Worker_i \stackrel{def}{=} Worker(\, execute_i\,)$, alors nous pouvons écrire un système composé de n processus travailleurs de la façon suivante

$$System \stackrel{def}{=} (\,\nu\, execute_i : i = 1, \ldots, n\,) \left(Worker_1 \mid \cdots \mid Worker_n \right) \tag{2.8}$$

Considérons le cas le plus simple, où le nombre de tâches, noté n, est égal au nombre de processus travailleurs disponibles. Nous attribuons à chaque agent $Worker_i$ (via son canal $execute_i$) la tâche $Task_i$ (avec $Task_i \equiv Task$). Le système s'enrichit et peut s'écrire comme suit

$$\begin{aligned} System \stackrel{def}{=}\ & (\,\nu\, execute_i : i = 1, \ldots, n\,) \\ & Worker_1 \mid \cdots \mid Worker_n \mid \overline{execute_1}\,\langle\, Task_1\,\rangle \mid \cdots \mid \overline{execute_n}\,\langle\, Task_n\,\rangle \end{aligned} \tag{2.9}$$

Chaque communication, à travers les canaux $execute_i$, permet à l'agent $Worker_i$ correspondant de recevoir un agent $Task_i$, il y a alors une substitution de sa variable libre T par l'agent $Task_i$

4. Nous utilisons différentes abréviations dans la suite de ce chapitre. Si leur utilisation permet une facilité d'écriture et de lecture, elle n'est en aucun cas une facilité de spécification.

reçu.

L'envoi des n agents $Task_1 \dots Task_n$ à travers les canaux $execute_1 \dots execute_n$ conduit à une réduction du terme $System$ (eq. 2.9) comme suit

$$System \xrightarrow{\overline{execute_i}\langle Task_i \rangle}{}^{n} Worker_1 \ \{^{Task_1}/_T\} \mid \cdots \mid Worker_n \ \{^{Task_n}/_T\} \tag{2.10}$$

Chaque agent $Task_i$ est alors évalué dans le contexte local du $Worker_i$ ayant reçu l'agent $Task_i$ via le canal $execute_i$.

2.2.1.2 Définition d'un processus « maître »

Dans le modèle *Maître-Travailleurs*, l'exécution du processus maître respecte le scénario suivant :

1. distribution des différentes tâches aux processus travailleurs participant à la résolution du cas de calcul (dans notre cas aux n agents $Worker$) ;
2. collecte des résultats de chaque tâche ;
3. finalisation du cas de calcul.

Pour définir le terme $Master$ qui modélise ce processus maître, nous faisons évoluer le terme $Task$ (eq. 2.5) modélisant une tâche de calcul. Le terme $Task$ (eq. 2.11) est maintenant paramétré par un canal $finish$, celui-ci permet à un agent $Task$ de signaler la fin de l'exécution de la tâche qu'il modélise et de transmettre le résultat (modélisé par le nom $result$). Cette évolution entraîne l'évolution de l'abréviation $Task_i$ telle que $Task_i \stackrel{def}{=} Task(\, finish_i \,)$.

$$Task(\, finish \,) \stackrel{def}{=} \tau \,.\, (\, \nu \, result \,) \, \overline{finish} \langle \, result \, \rangle \,.\, 0 \tag{2.11}$$

Le terme $Worker$ (eq. 2.12) évolue de concert avec le terme $Task$ (eq. 2.28) afin de prendre en compte le nouveau paramètre $finish$ (eq. 2.11).

$$Worker(\, execute \,) \stackrel{def}{=} execute(\, T(\, finish \,) \,) \,.\, T(\, finish \,) \tag{2.12}$$

Pour modéliser le processus maître, nous définissons le terme $Master$ (eq. 2.13). Ce terme est paramétrable par le vecteur $\vec{execute}_n$. Ce vecteur, composé de n canaux notés $execute_1 \dots execute_n$, permet à un agent $Master$ d'envoyer à chaque agent $Worker_i$ un agent $Task_i$. Nous pouvons traduire cette notation par « *le processus maître affecte à chaque processus travailleur une tâche* ».

$$\begin{aligned} Master(\, \vec{execute}_n \,) \stackrel{def}{=} \ & (\, \nu \, finish_i : i = 1, \dots, n \,) \\ & \left(\begin{array}{cc} \overline{execute_1} \langle \, Task_1 \, \rangle \,.\, finish_1(\, result_1 \,) \\ \vdots \qquad\qquad \vdots \\ \mid \quad \overline{execute_n} \langle \, Task_n \, \rangle \,.\, finish_n(\, result_n \,) \end{array} \right) . \, \tau \,.\, 0 \end{aligned} \tag{2.13}$$

Un agent $Master$ s'ajoute aux n agents $Worker_i$ déjà présents dans le système (2.8) , le système est maintenant défini comme suit

$$System_1 \stackrel{def}{=} (\nu\, execute_i : i = 1, \ldots, n\,) \Big(Master(\vec{execute}_n\,) \mid Worker_1 \mid \cdots \mid Worker_n \Big) \tag{2.14}$$

L'agent $Master$ rythme la résolution du cas de calcul, celle-ci peut être décomposée en 3 étapes successives, que nous listons ci-après :

a) **Affectation des tâches**

Le processus maître, défini par l'agent $Master$, distribue les n tâches (modélisées par les agents $Task_1 \ldots\ Task_n$) aux n processus travailleurs (modélisés par les agents $Worker_1$ $\ldots\ Worker_n$). Cette suite d'actions est modélisée par les n réductions (eq. 2.15) du terme $Master$ par n transitions $\overline{execute_i}\,\langle Task_i \rangle$ avec $i = 1, \ldots, n$.

$$Master(\vec{execute}_n\,) \xrightarrow{\overline{execute_i}\,\langle Task_i \rangle}{}^{n} \Big(finish_1(result) \mid \cdots \mid finish_n(result) \Big).\ \tau\ .\ 0 \tag{2.15}$$

Les n réductions duales (eq. 2.16) sont réalisées par les n transitions $execute_i(Task_i)$ définies pour chaque $Worker_i$. Cela signifie qu'un agent $Worker_i$ reçoit un agent $Task_i$ via son canal $execute_i$ et continue son exécution comme $Task_i$, avec $1 \leq i \leq n$.

$$Worker_1 \mid \cdots \mid Worker_n \xrightarrow{execute_i(Task_i)}{}^{n} \begin{array}{l} \Big(\tau\ .\ (\nu\, result_1)\ \overline{finish_1}\,\langle result_1 \rangle\ .\ 0\ \Big) \\ \quad\vdots \qquad\qquad \vdots \\ \Big(\tau\ .\ (\nu\, result_n)\ \overline{finish_n}\,\langle result_n \rangle\ .\ 0\ \Big) \end{array} \tag{2.16}$$

b) **Exécution des tâches**

L'affectation d'une tâche à chaque processus travailleur, modélisé par un agent $Worker_i$, entraine le système dans l'état $System_2$ décrit par le terme suivant

$$System_2 \equiv \underbrace{\Big((finish_1(result_1) \mid \cdots \mid finish_n(result_n)\ ).\ \tau\ .\ 0\Big)}_{Master} \mid \underbrace{\Big(\tau\ .\ (\nu\, result_1)\ \overline{finish_1}\,\langle result_1 \rangle\ .\ 0\ \Big)}_{Worker_1} \mid \cdots \mid \underbrace{\Big(\tau\ .\ (\nu\, result_n)\ \overline{finish_n}\,\langle result_n \rangle\ .\ 0\ \Big)}_{Worker_n} \tag{2.17}$$

Chaque agent $Worker_i$ exécute localement le code (représenté par τ) contenu dans l'agent $Task_i$ qu'il a reçu par son canal $execute_i$.

c) **Collecte des résultats et finalisation du calcul**
La fin du cas de calcul correspond avec la fin de l'exécution de l'agent $Master$. La collecte des résultats (via les canaux $finish_1 \ \ldots \ finish_n$) est réalisée, l'agent $Master$ est équivalent à τ (ce qui représente un traitement local fait par le processus maître).

2.2.1.3 Bilan

Cette première approche du π-calcul pour modéliser le modèle de programmation parallèle *Maître-Travailleurs* nous a permis de définir les trois termes $Master$ (eq. 2.13), $Worker$ (eq. 2.12) et $Task$ (eq. 2.11). Ces termes sont présents dans toutes les architectures logicielles que nous modélisons dans la suite de ce chapitre. Enfin, l'utilisation du π-calcul d'ordre supérieur, nous a permis de modéliser un agent mobile avec le terme $Task$. Ce terme apporte le premier aspect de l'« adaptabilité » recherchée pour notre architecture logicielle.

2.2.2 Modélisation d'une architecture adaptable

L'exécution d'un code suivant le modèle *Maître-Travailleurs* est exposée à de fortes contraintes comme :

– le nombre de processus participant au calcul doit être défini avant de commencer le calcul ;
– si un processus participant au calcul n'est plus disponible alors le calcul est bloqué ;
– le nombre de tâches est fortement dépendant du nombre de processus définis.

La résolution d'un cas de calcul dans l'architecture logicielle que nous avons précédemment définie (eq. 2.14) peut conduire à un état bloquant. En effet, l'exécution d'une tâche est réalisée après son affectation à l'un des agents $Worker$ présents dans le système. Si le nombre d'agents $Worker$ n'est pas égal ou supérieur au nombre d'agents $Task_i$ alors le calcul est bloqué. Dans ce cas, le calcul se trouverait bloqué au moment de l'affectation d'une tâche par l'agent $Master$ à l'agent $Worker$ manquant. Ceci ne respecterait pas la *transparence d'échelle*.

Nous définissons dans cette section une architecture logicielle capable de répondre à ce besoin de disponibilité maximale. Afin d'éviter la situation dans laquelle la résolution d'un cas de calcul serait bloquée, l'agent $Master$ ne va plus communiquer directement avec les agents $Worker$. Nous avons choisi d'utiliser pour cela un *annuaire de tâches* pour que l'agent $Master$ dépose les tâches de calcul et que les agents $Worker$ viennent les récupérer. De ce fait, le processus maître n'est plus dépendant du nombre de processus travailleurs qui participent au cas de calcul. Notons aussi que les tâches que nous définissons sont indépendantes, c'est à dire qu'aucune tâche n'a besoin du résultat d'une autre pour continuer son calcul. Ceci évite d'avoir un cas de calcul bloqué au niveau de ses processus travailleurs. En effet, dans le cas contraire, une tâche affectée à un processus travailleur pourrait se retrouver dans l'attente du résultat d'une tâche non affectée.

2.2.2.1 La modélisation d'un « code mobile »

Avant de modéliser l'annuaire de tâches, une évolution du terme $Task$ (eq. 2.11) est nécessaire. En effet, dans cette nouvelle architecture, un agent *Task* n'est plus envoyé directement à un agent $Worker$. C'est maintenant un agent $Worker$ qui demande à la tâche le code qu'il doit exécuter. Le terme $Task$ (eq. 2.18) ne définit plus un agent mobile mais le conteneur d'un agent mobile. Cet agent mobile est le code à exécuter pour la tâche et il est défini par le terme $ComputeAgent$ (eq. 2.19).

$$\begin{aligned} Task(\, getTask, finish\,) \stackrel{def}{=}\ & getTask(\, worker\,) \\ & .\ (\,\nu\, ack_{finish}\,)\ \Big(\overline{worker}\,\langle\, ComputeAgent(\, ack_{finish}\,)\ \rangle\ .\ ack_{finish}(\, result\,)\ \Big) \\ & .\ \overline{finish}\,\langle\, result\,\rangle\ .\ 0 \end{aligned} \tag{2.18}$$

avec

$$ComputeAgent(\, finish\,) \stackrel{def}{=}\ \tau\ .\ (\,\nu\, result\,)\ \overline{finish}\,\langle\, result\,\rangle\ .\ 0 \tag{2.19}$$

Le terme $Task$ envoie le code à exécuter (modélisé par un agent défini par le terme $ComputeAgent$) à travers le canal $worker$ qu'il a reçu lors de la demande de la tâche réalisée à travers le canal $getTask$. Le terme $ComputeAgent$ est similaire au terme $Task$ de l'équation 2.11, il est donc uniquement paramétrable par le nom $finish$. Ce canal permet ensuite à un agent défini par le terme $ComputeAgent$ d'informer l'agent $Task$ correspondant de la fin de l'exécution du code. Le terme $Task$ (eq. 2.18) est quant à lui paramétré par les canaux $getTask$ et $finish$. Le canal $getTask$ permet à un agent $Worker$ (eq. 2.28) de communiquer un canal $worker$ à travers lequel un agent $Task$ peut lui envoyer son agent $ComputeAgent$ correspondant.

L'évolution du terme $Task$ (eq. 2.18) entraine la modification de l'abréviation $Task_i$ telle que

Notons que les agents $Task_i$, avec $i = 1, \ldots, n$ (où n est le nombre de tâches définies pour un cas de calcul), sont tous paramétrés par le même canal $getTask$. Cette notation n'est pas une abréviation pour simplifier l'écriture mais elle indique que le canal $getTask$ est partagé par tous les agents définis par $Task$. Un agent $Worker$ n'est donc pas lié à un agent $Task_i$ en particulier avant de recevoir un agent défini par le terme $ComputeAgent$ par le canal $worker$.

2.2.2.2 Définition d'un annuaire de tâches

L'annuaire de tâches est défini par le terme $TaskDirectory$ (eq. 2.20)

$$TaskDirectory(\, add\,) \stackrel{def}{=}\ !\ \Big(add(\, T(\, g, f\,)\ )\ .\ T(\, g, f\,)\ \Big) \tag{2.20}$$

L'ajout d'un agent $Task_i$ dans l'annuaire passe par l'action $add(\,Task_i\,)$. Cette action conduit à la réduction du terme $TaskDirectory$ comme suit

$$TaskDirectory(\,add\,) \xrightarrow{add(\,Task_i\,)} !\,\Big(add(\,T(\,g,f\,)\,)\,.\,T(\,g,f\,)\,\Big)\big|\,Task_i \tag{2.21}$$

Pour utiliser un annuaire de tâches, le terme $Master$ (eq. 2.13) doit évoluer. En effet, un agent défini par le terme $Master$ (eq. 2.22) ne communique plus avec les agents $Worker$ mais avec l'agent $TaskDirectory$.

$$\begin{aligned} Master(\,getTask, add_T\,) \stackrel{def}{=}\ & (\,\nu\, finish_i : i = 1,\ldots,n\,) \\ & \left(\begin{array}{cc} & \overline{add_T}\,\langle\, Task_1\,\rangle\,.\, finish_1(\,result_1\,) \\ \big| & \vdots \qquad\qquad \vdots \\ \big| & \overline{add_T}\,\langle\, Task_n\,\rangle\,.\, finish_n(\,result_n\,) \end{array}\right).\,\tau\,.\,0 \end{aligned} \tag{2.22}$$

Le vecteur $\vec{execute}$ a disparu et est remplacé par deux nouveaux paramètres : les canaux add_T et $getTask$. Le terme $Master$ n'a plus de lien direct avec les agents $Worker$ mais ajoute les tâches dans l'annuaire à travers le canal add_T. Le canal $getTask$, commun à tous les agents $Task_i$ (eq. **??**), permet aux agents $Worker$ de communiquer indépendamment avec tous les agents $Task_i$. Le système (eq. 2.23) est maintenant constitué de deux agents : un agent défini par le terme $Master$ et un agent défini par le terme $TaskDirectory$.

$$\begin{aligned} System \stackrel{def}{=}\ & (\,\nu\, add_T, getTask\,) \\ & \Big(Master(\,getTask, add_T\,)\ \big|\ TaskDirectory(\,add_T\,)\,\Big) \end{aligned} \tag{2.23}$$

A travers le canal add_T, l'agent $Master$ ajoute les n tâches $(Task_1 \ldots Task_n)$ à l'agent $TaskDirectory$. Les n actions $add_T(\,Task_i\,)$ (avec $i = 1, \ldots, n$) modélisent la réception de ces n agents par l'agent $TaskDirectory$. Notons $TaskDirectory'$ la réduction de l'agent $TaskDirectory$ suite à ces n actions (eq. 2.24).

$$\begin{aligned} TaskDirectory(\,add_T\,) & \xrightarrow{add_T(\,Task_i\,)}{}^{n}\ !\,\Big(add(\,T(\,g,f\,)\,)\,.\,T(\,g,f\,)\,\Big)\big|\,Task_1\ \big|\ \cdots\ \big|\ Task_n \\ & \xrightarrow{add_T(\,Task_i\,)}{}^{n}\ TaskDirectory'(\,add_T\,) \end{aligned} \tag{2.24}$$

Une réduction duale de l'agent $Master$ est faite par n actions $\overline{add_T}\,\langle\, Task_i\,\rangle$ avec $i = 1, \ldots, n$. Notons $Master'$ la réduction de l'agent $Master$ après ces n actions (eq. 2.25).

$$\begin{aligned} Master(\,getTask, add_T\,) & \xrightarrow{\overline{add_T}\,\langle\, Task_i\,\rangle}{}^{n}\ \Big(finish_1(\,result_1\,)\ \big|\ \cdots\ \big|\ finish_n(\,result_n\,)\,\Big).\,\tau\,.\,0 \\ & \xrightarrow{\overline{add_T}\,\langle\, Task_i\,\rangle}{}^{n}\ Master'(\,getTask, add_T\,) \end{aligned} \tag{2.25}$$

L'ajout des n tâches à l'annuaire conduit le système dans l'état suivant, noté $System'$:

$$\begin{aligned} System' &\stackrel{def}{=} (\nu\, add_T, getTask)\left(Master'(\,getTask, add_T\,) \mid TaskDirectory'(\,add_T\,)\right) \\ &\stackrel{def}{=} \left(\begin{array}{l} Master'(\,getTask, add_T\,) \mid !\left(add(\,T(\,g,f\,)\,)\,.\,T(\,g,f\,)\right) \\ \mid \underbrace{getTask(\,worker\,)\,.\,\cdots \mid \cdots \mid getTask(\,worker\,)\,.\,\cdots}_{n \text{ fois}} \end{array} \right) \end{aligned} \tag{2.26}$$

Après l'ajout des tâches dans l'annuaire, le système (eq. 2.26) est bloqué. Les n agents $Task_i$ sont dans l'attente d'une réception sur le canal $getTask$. Cette situation représente les n tâches en attente de traitement [5]. Le système continue son exécution lorsqu'une action d'émission à travers le canal $getTask$ est réalisée. Cette action est réalisée par les agents $Worker$ que nous définissons et ajoutons au système dans la section.

Remarque : Il est possible de définir $Task_i \stackrel{def}{=} Task(\,getTask, finish\,)$. Dans ce cas, le canal $finish$ devient, à l'instar du canal $getTask$, commun à tous les agents $Task$. Cependant, paramétrer tous les agents $Task$ avec le même canal $finish$ n'est pas une contrainte de notre modélisation, contrairement au paramètre $getTask$. Ce cas particulier peut être défini lors d'un cas de calcul qui n'a pas de contrainte sur l'ordre d'exécution des tâches.
Dans ce cas particulier, la réduction faite par l'ajout des n agents $Task_i$ à travers le canal add_T est modélisée de la façon suivante

$$Master(\,getTask, add_T\,) \xrightarrow{\overline{add_T}\,\langle\, Task_i\,\rangle}{}^{n} \underbrace{\left(finish(\,result_1\,) \mid \cdots \mid finish(\,result_n\,)\right)}_{n \text{ fois}}.\,\tau\,.\,0 \tag{2.27}$$

2.2.2.3 Des processus travailleurs tous identiques

Nous faisons évoluer le terme $Worker$ (eq. 2.28) pour qu'il puisse communiquer à travers le canal $getTask$ commun à tous les agents $Task_i$ (eq. 2.26). Le terme $Worker$ est maintenant paramétré par le canal $getTask$ et peut exécuter plusieurs tâches à la suite grâce à l'utilisation de la récursion. Le canal $task$ est alors le support de la réception d'un agent mobile ayant une signature particulière, c'est à dire avec un paramètre formel (ici noté f).

$$\begin{aligned} Worker(\,getTask\,) &\stackrel{def}{=} (\nu\, task)\left(\overline{getTask}\,\langle\, task\,\rangle\,.\,task(\,CA(\,f\,)\,)\right) \\ &\quad .\,CA(\,f\,)\,.\,Worker(\,getTask\,) \end{aligned} \tag{2.28}$$

5. Il est à noter que nous pourrions aussi dire que le système est bloqué sur les canaux $finish_1 \ldots finish_n$ présents dans l'agent $Master$ (eq. 2.25). Cependant, une action d'émission via un canal $finish_i$ (faite par chaque agent $Task_i$ correspondant) est la conséquence logique d'une action de réception sur un canal $getTask$, il est donc inutile de dire que le système, dans l'état actuel, est bloqué sur les canaux $finish_1 \ldots finish_n$.

Nous ajoutons m agents $Worker$ au système précédent (eq. 2.29) pour répondre aux n actions de réception $getTask(worker)$ définies par les agents $Task_i$. Il y a donc, à présent, m processus travailleurs pour traiter n tâches de calcul. Nous rappelons que $Master'$ (eq. 2.25) et $TaskDirectory'$ (eq. 2.24) représentent respectivement les réductions des termes $Master$ et $TaskDirectory$ après l'ajout des n tâches dans l'annuaire.

$$System \equiv (\nu\, add_T, getTask) \left(\begin{array}{l} Master'(getTask, add_T) \mid TaskDirectory'(add_T) \\ \mid \underbrace{Worker(getTask) \mid \cdots \mid Worker(getTask)}_{m \text{ fois}} \end{array} \right) \tag{2.29}$$

Tous les agents $Worker$ du système sont paramétrés par le même canal $getTask$. Ce canal est aussi le paramètre de tous les agents $Task_i$ du cas de calcul qui sont présents dans l'annuaire ($TaskDirectory'$ (eq. 2.24)) .

$$System \equiv (\nu\, add_T, getTask) \left(\begin{array}{l} Master'(getTask, add_T) \mid \underbrace{getTask(worker).\cdots \mid \cdots \mid getTask(worker).\cdots}_{n \text{ fois} = n \text{ agents } Task} \\ \mid \underbrace{(\nu\, task)\, \overline{getTask}\langle task\rangle.\cdots \mid \cdots \mid (\nu\, task)\, \overline{getTask}\langle task\rangle.\cdots}_{m \text{ fois} = m \text{ agents } Worker} \end{array} \right) \tag{2.30}$$

Les m agents $Worker$ ont donc tous la possibilité de communiquer avec les n agents $Task_i$. En effet, d'un coté, tous les agents $Worker$ sont identiques et la première action de chacun des agents $Worker$ est une émission à travers le canal $getTask$. D'un autre coté, les agents $Task_i$ sont tous différents mais attendent tous une action de réception sur ce même canal $getTask$. Un agent $Worker$ communique donc indifféremment avec un agent $Task_i$ (avec $1 < i \leq n$) à travers le canal $getTask$. L'arrivée de m agents $Worker$ dans le système est modélisée par l'abréviation $(\ \overline{getTask}\ .\ \ldots\ \mid\ )^m$. L'abréviation $(\ getTask\ .\ \ldots\ \mid\ )^n$ modélise les n tâches en attente de traitement.

$$System \stackrel{def}{=} \left(\begin{array}{l} Master'(getTask, add_T) \mid \overbrace{\cdots \mid \underbrace{\left(getTask(worker).\cdots \mid \right)^n}_{n \text{ agents } Task_i}}^{TaskDirectory'} \\ \mid \underbrace{\left((\nu\, task)\, \overline{getTask}\langle task\rangle.\ \ldots\ \mid \right)^m}_{m \text{ agents } Worker} \end{array} \right) \tag{2.31}$$

2.2.2.4 L'exécution des tâches

La réduction (eq. 2.32) du terme $System$ (eq. 2.31), notée $\xrightarrow{\tau}$, modélise la demande d'une tâche par un agent $Worker$

$$
\begin{array}{l}
System \xrightarrow{\tau} (\,\nu\, add_T, getTask\,) \\
\left(\begin{array}{l}
Master'(\,getTask, add_T\,) \\
\mid \underbrace{\Big(\; getTask(\,worker\,)\;.\;\ldots \mid \Big)^{n-1}}_{n \text{ agents } Task} \mid \Big(\; \overline{task}\,\langle\, CA_j \,\rangle\;.\;\cdots \Big) \\
\mid \underbrace{\Big(\; (\,\nu\, task\,)\;\overline{getTask}\,\langle\, task \,\rangle\;.\;\ldots \mid \Big)^{m-1}}_{m \text{ agents } Worker} \mid \Big(\; task(\,CA_j\,)\;.\;\cdots \Big)
\end{array} \right)
\end{array}
\tag{2.32}
$$

Un agent $Worker$ transmet un canal $task$ à un agent $Task_j$ (avec $1 \leq j \leq n$). Ce canal permet à l'agent $Task_j$ de communiquer l'agent mobile CA_j, avec $CA_j = ComputeAgent(f_j)$, à cet agent $Worker$. L'agent $Worker$ devient alors spécifique à cette tâche (représentée par l'agent $Task_j$) car il exécute le code (ici représenté par l'action interne τ) défini par l'agent CA_j.

L'équation 2.33 modélise les réductions successives au couple d'agents $Task_j$ (eq. 2.18) et $Worker$ (eq. 2.28) lors du traitement d'une tâche (agent $Task_j$) par un processus travailleur (agent $Worker$).

$$
\begin{array}{ll}
 & Task(\,getTask, finish_j\,) \mid Worker(\,getTask\,) \\
\xrightarrow{\tau}\xrightarrow{\tau} & \Big(\; f_j(\,result_j\,)\;.\;\overline{finish_j}\,\langle\, result_j \,\rangle\;.\;0\;\Big) \mid \Big(\; CA_j\;.\;Worker(\,getTask\,)\;\Big) \\
\xrightarrow{\tau}\xrightarrow{\tau} & \Big(\; \overline{finish_j}\,\langle\, result_j \,\rangle\;.\;0\;\Big) \mid Worker(\,getTask\,)
\end{array}
\tag{2.33}
$$

Les deux premières réductions permettent à l'agent $Worker$ de devenir spécifique à la tâche définie par l'agent $Task_j$. En effet, l'agent $Worker$ est prêt à exécuter le code défini par l'agent mobile CA_j.

Les deux réductions suivantes modélisent l'exécution de l'agent CA_j et la réception de son résultat ($result_j$) à travers le canal f_j par l'agent $Task_j$.

Notons α le nombre de réductions tel que

$$
Worker \mid Task_j \xrightarrow{\tau}{}^{\alpha} Worker \mid \overline{finish_j}\,\langle\, result_j \,\rangle
\tag{2.34}
$$

nous pouvons dire que α est le nombre de réductions nécessaires pour l'exécution d'une tâche par un agent $Worker$.

Dans l'équation 2.31, le nombre n de tâches est indépendant du nombre m de processus travailleurs disponibles. Nous pouvons donc lister les trois cas suivants :
– $\mathbf{m = n}$ alors le nombre de processus travailleurs disponibles est identique au nombre de

tâches ;
– **m** $>$ **n** alors le nombre de processus travailleurs disponibles est supérieur au nombre de tâches ;
– **m** $<$ **n** alors le nombre de processus travailleurs disponibles est inférieur au nombre de tâches.

Les trois cas conduisent tous à la même réduction du terme $System$ (eq. 2.33) après $n \times \alpha$ réductions. Tous les agents $Worker$ sont identiques au début de leur exécution et c'est à la réception d'un agent mobile défini par le terme CA qu'un agent $Worker$ se différencie des autres. Il exécute alors le code correspondant à cet agent défini par le terme CA puis reprend une exécution semblable à tous les autres agents $Worker$ (grâce à l'utilisation de la récursion).

$$
\begin{aligned}
&System \xrightarrow{\tau}{}^{n\times\alpha} \\
&\quad \left(\overbrace{\left(\left(finish_1(result_1) \mid \cdots \mid finish_n(result_n) \right).\, \tau \,.\, 0 \right)}^{Master} \right. \\
&\quad \mid \overbrace{\left(\underbrace{\left(\overline{finish_1}\langle result_1 \rangle \mid \cdots \mid \overline{finish_n}\langle result_n \rangle \right)}_{n \text{ agents } Task_i} \mid \cdots \right)}^{TaskDirectory} \\
&\quad \left. \mid \underbrace{Worker(getTask) \mid \cdots \mid Worker(getTask)}_{m \text{ agents } Worker} \right) \\
&System \xrightarrow{\tau}{}^{n\times\alpha} System'
\end{aligned}
\tag{2.35}
$$

Dans $System'$ (eq. 2.35), toutes les tâches ont été traitées par les agents $Worker$. Les $n + 1$ réductions suivantes entre les agents $Task_i$ et l'agent $Master$ finalisent le cas de calcul (eq. 2.36) avec :
– n communications à travers les canaux $finish_i$ pour que l'agent $Master$ récupère les résultats ;
– et une action silencieuse (dans $Master$) pour finaliser le cas de calcul.

$$
\begin{aligned}
&System' \xrightarrow{\tau}{}^{n\,+\,1} \\
&\quad TaskDirectory \mid \underbrace{Worker(getTask) \mid \cdots \mid Worker(getTask)}_{m \text{ agents } Worker}
\end{aligned}
\tag{2.36}
$$

2.2.2.5 Définition d'une « ferme de travailleurs »

Nous définissons une « ferme de travailleurs » comme étant un ensemble d'agents « identiques », tous définis par le terme $Worker$ (eq. 2.28). Tous ces agents sont paramétrés par le même canal nommé $getTask$ et sont capables d'exécuter toutes les tâches. Un agent $Worker$ s'adapte à une tâche, modélisée par un agent $Task_j$, grâce à l'agent mobile CA_j correspondant à la tâche. A la fin du traitement d'une tâche, un agent $Worker$ redevient identique aux autres agents $Worker$ de la ferme.

Le terme $Worker^m$ (eq. 2.37) définit une ferme de m « travailleurs » :

$$Worker^m(\, getTask\,) \stackrel{def}{=} \underbrace{Worker(\, getTask\,) \mid \cdots \mid Worker(\, getTask\,)}_{m \text{ fois}} \tag{2.37}$$

2.2.2.6 Bilan

La modélisation d'une architecture logicielle suivant le modèle *Maître-Travailleurs* réalisée dans la section 2.2.1 a évolué. Le système (eq. 2.38) est maintenant composé de 3 éléments principaux, que nous rappelons :
- un processus maître, modélisé par le terme $Master$ (eq. 2.22),
- un annuaire de tâches, modélisé par le terme $TaskDirectory$ (eq. 2.20),
- une « ferme de travailleurs », modélisée par le terme $Worker^m$ (eq. 2.37),

$$\begin{aligned} System \stackrel{def}{=}\ & (\,\nu\, add_T, getTask\,) \\ & \Big(Master(\, getTask, add_T\,) \mid TaskDirectory(\, add_T\,) \mid Worker^m(\, getTask\,)\,\Big) \end{aligned} \tag{2.38}$$

L'évolution de notre spécification a conduit à la prise en compte de deux transparences présentées dans la section 1.1.2 :
- la *transparence de localisation* : les agents mobiles définis par le terme CA sont exécutés par des agents $Worker$ dont la localisation n'est pas connue,
- la *transparence d'échelle* : le nombre d'agents $Worker$ est désormais indépendant du nombre d'agents $Task_i$.

Nous pouvons noter qu'après l'évaluation du cas de calcul, représenté par l'agent $Master$, le système est prêt pour l'évaluation d'un nouveau cas de calcul qui serait modélisé par un autre agent $Master$.

2.2.3 Modélisation d'un cas de calcul

Dans cette section, nous continuons l'écriture de notre spécification π-calcul pour arriver à la modélisation d'un cas de calcul. Nous commençons par ajouter la définition de termes pour modéliser l'accès à des données (Section 2.2.3.1) et la définition de propriétés globales (Section 2.2.3.1) dans un cas de calcul. Ensuite, nous regroupons ces termes pour définir un cas de calcul de manière générale (Section 2.2.3.4). Nous terminons par modéliser un exemple de cas de calcul (Section 2.2.3.5).

2.2.3.1 Modélisation d'une mémoire distribuée

Dans le but de garder une *transparence de localisation* vis-à-vis des données sur la grille de calcul (cf. Section 1.1.2), l'accès aux données physiques se fait par un gestionnaire de données. Ce

gestionnaire fournit des liens vers des blocs de données pour les lire et/ou les modifier. L'accès à un bloc de données est modélisé par le terme $DataHandler$. Ce terme est paramétrable par les canaux $read$ et $write$ et l'action τ représente l'accès à la ressource physique.

$$\begin{aligned} DataHandler(\,read, write\,) \stackrel{def}{=} \\ & \Big(read(\,upload\,)\ .\ \tau\ .\ (\,\nu\, data\,)\ \overline{upload}\,\langle\, data\,\rangle\ .\ DataHandler(\,read, write\,)\ \Big) \\ + \quad & \Big(write(\,data\,)\ .\ \tau\ .\ DataHandler(\,read, write\,)\ \Big) \end{aligned} \tag{2.39}$$

Dans un but similaire à celui qui a conduit à la définition du terme $TaskDirectory$ (eq. 2.20), c'est à dire rendre disponible les agents définis par le terme $Task$, nous définissons le terme $DHDirectory$ (eq. 2.40). Celui-ci définit un annuaire mettant à disposition des agents définis par le terme $DataHandler$ à tous les agents (définis par les termes $Worker$ et $Master$) participant à la résolution du cas de calcul. Ce terme est paramétrable par le canal *add*.

$$DHDirectory(\,add\,) \stackrel{def}{=} !\ \Big(\ add(\,DH(\,r, w\,)\ )\ .\ DH(\,r, w\,)\ \Big) \tag{2.40}$$

L'agent $DHDirectory$ est le gestionnaire de données de notre architecture car il permet l'accès aux données à tous les processus qui la composent.

2.2.3.2 Communications entre les données et les tâches

Lors d'un cas de calcul suivant le modèle SPMD, chaque tâche exécute le même code mais sur des blocs de données différents. Dans notre architecture, le code de chaque tâche est modélisé par l'action interne τ défini dans le terme $ComputeAgent$ (eq. 2.19). Pour permettre aux agents mobiles définis par le terme $ComputeAgent$ d'accéder aux données, nous faisons évoluer ce terme (eq. 2.41). Celui-ci introduit le canal *init* comme paramètre. Par ce canal *init*, un agent défini par $ComputeAgent$ reçoit les canaux *input* et *output* permettant de communiquer avec un agent défini par le terme $DataHandler$. Ces canaux sont fournis par la définition du terme $Task$ (eq. 2.42) dans le but d'accéder aux données spécifiques de la tâche.

$$\begin{aligned} ComputeAgent(\,\boldsymbol{init}, finish\,) \stackrel{def}{=}\ & \boldsymbol{init(\,input, output\,)} \\ & .\ (\,\nu\, download\,)\ \Big(\overline{input}\,\langle\, download\,\rangle\ .\ download(\,data\,)\ \Big) \\ & .\ \tau\ .\ (\,\nu\, newdata\,)\ \overline{output}\,\langle\, newdata\,\rangle\ .\ (\,\nu\, result\,)\ \overline{finish}\,\langle\, result\,\rangle\ .\ 0 \end{aligned} \tag{2.41}$$

Les canaux *input* et *output* permettent à un agent mobile défini par le terme $ComputeAagent$ de :

– de lire le bloc de données (*data*) avec

$$(\,\nu\, download\,)\,\left(\overline{input}\,\langle\, download\,\rangle\;.\; download(\,data\,)\;\right)$$

– de modifier le bloc de données, partagés dans l'annuaire, avec $\overline{output}\,\langle\, newdata\,\rangle$

L'évolution du terme *ComputeAgent* (eq. 2.41) impose de faire évoluer le terme $Task$ (eq. 2.18). Comme pour le terme *ComputeAgent*, le terme $Task$ introduit un canal *init* comme paramètre. Celui-ci lui permet de recevoir les canaux *input* et *ouput* de l'agent *DataHandler* correspondant aux données spécifiques à la tâche. Après avoir émis un agent défini par le terme *ComputeAgent* à travers le canal *worker*, le canal $init_{agent}$ lui permet d'initialiser cet agent mobile avec les canaux *input* et *output*.

$$\begin{aligned} &Task(\,init, getTask, finish\,) \stackrel{def}{=} \\ &\quad init(\,input, ouput\,)\;.\; getTask(\,worker\,) \\ &\quad .\left(\begin{array}{ll} \nu & init_{agent}, \\ & ack_{finish} \end{array}\right)\left(\begin{array}{l} \overline{worker}\,\langle\, ComputeAgent(init_{agent}, ack_{finish}\,\rangle \\ .\;\overline{init_{agent}}\,\langle\, input, output\,\rangle\;.\; ack_{finish}(\,result\,) \end{array}\right) \\ &\quad .\,\overline{finish}\,\langle\, result\,\rangle\;.\; 0 \end{aligned} \tag{2.42}$$

Nous modifions l'abréviation $Task_i$ (eq. **??**) afin de respecter l'évolution du terme $Task$ (eq. 2.42) et l'ajout du paramètre *init*. Chaque agent $Task_i$ (eq. 2.43) est maintenant paramétré par trois canaux. Les canaux $getTask$ et $finish$ gardant les rôles identiques décrits dans la section 2.2.2.1.

$$Task_i \stackrel{def}{=} Task(\,init_i, getTask, finish_i\,) \tag{2.43}$$

2.2.3.3 La répartition des données

Nous continuons à faire évoluer notre spécification avec la modélisation du processus maître. Dans un cas de calcul suivant le modèle SPMD, celui-ci met à disposition des blocs de données aux processus travailleurs et définit une tâche par bloc. Le terme $Master$ (eq. 2.22) évolue afin de pouvoir communiquer avec un agent défini par le terme *DataHandler* et invoquer des agents définis par le terme $Task$ (eq. 2.42). Le nouveau terme $Master$ possède un paramètre add_{DH} en plus qui permet de récupérer un canal vers un agent défini par le terme *DataHandler*. Les n agents $TaskHandler_i$ (eq. 2.45) (avec $i = 1, \ldots, n$) modélisent l'ajout des n tâches et les accès aux blocs de données correspondants.

$$\begin{aligned} &Master(\,getTask, add_T, add_{DH}\,) \stackrel{def}{=} \\ &\quad \left(\begin{array}{l} TaskHandler_1(\,getTask, add_T, add_{DH}\,) \\ \qquad \mid \cdots \mid TaskHandler_n(\,getTask, add_T, add_{DH}\,) \end{array}\right).\,\tau\;.\; 0 \end{aligned} \tag{2.44}$$

avec

$$TaskHandler_i(\, add_T, add_{DH}, getTask\,) \stackrel{def}{=} (\,\nu\, read_i, write_i, init_i, finish_i\,) \left(\begin{array}{l} \overline{add_{DH}}\,\langle\, DH_i\,\rangle\, .\, \overline{add_T}\,\langle\, Task_i\,\rangle\, .\, \overline{init_i}\,\langle\, read_i, write_i\,\rangle\, .\, finish_i(\, result_i\,) \\ .\, (\,\nu\, download\,)\, \left(\overline{read_i}\,\langle\, download\,\rangle\, .\, download(\, data_i\,)\,\right) \end{array} \right) \tag{2.45}$$

et

$$DH_i \stackrel{def}{=} DataHandler(\, read_i, write_i\,) \tag{2.46}$$

L'exécution des tâches n'est pas ordonnée, le processus maître ajoute les n tâches en parallèle. Pour chaque tâche, modélisée par un agent $Task_i$ (eq. 2.43), un agent $TaskHandler_i$ (avec $i = 1, \ldots, n$) :

- crée $((\,\nu\, read_i, write_i\,)\,)$ et partage $(\overline{add_{DH}}\,\langle\, DH_i\,\rangle\,)$ le bloc de données, via son gestionnaire DH_i (eq. 2.46), traité par la tâche ;
- crée $((\,\nu\, init_i, finish_i\,)\,)$ et ajoute $(\overline{add_T}\,\langle\, Task_i\,\rangle\,)$ la tâche à l'annuaire $TaskDirectory$;
- initialise l'agent $Task_i$ avec l'action $\overline{init_i}\,\langle\, read_i, write_i\,\rangle$;
- attend la fin du traitement de la tâche à travers le canal $finish_i$;
- lit le bloc de données ($data_i$) modifié lors de l'exécution de la tâche par la suite d'actions $(\,\nu\, download\,)\, \left(\overline{read_i}\,\langle\, download\,\rangle\, .\, download(\, data_i\,)\,\right)$.

Cette mémoire, partagée entre le processus maître et les processus travailleurs, est ajoutée au système sous le nom de $DHDirectory$ (eq. 2.47) et l'agent $Master$ communique avec l'agent $DHDirectory$ à travers le canal add_{DH}.

$$System \stackrel{def}{=} (\,\nu\, add_T, add_{DH}, getTask\,) \left(\begin{array}{l} TaskDirectory(\, add_T\,) \mid \boldsymbol{DHDirectory(\, add_{DH}\,)} \\ \mid Master(\, getTask, add_T, \boldsymbol{add_{DH}}\,) \mid Worker^m(\, getTask\,) \end{array} \right) \tag{2.47}$$

2.2.3.4 Définition d'un cas de calcul

Un cas de calcul possède généralement des propriétés de niveau global telles qu'une borne temporelle, un système d'unité ou parfois même des uri de ressources. Dans notre architecture, un processus maître et les processus travailleurs qui participent à un cas de calcul ont la possibilité de partager ces propriétés « globales ». Le terme $Property$ (eq. 2.48) modélise ce type de

propriété.

$$\begin{aligned} Property(\, value, read, write\,) \stackrel{def}{=} \\ \Big(read(\, reader\,) \,.\, \overline{reader}\,\langle\, value\,\rangle \,.\, Property(\, value, read, write\,)\,\Big) \\ + \quad \Big(write(\, value_{new}\,) \,.\, Property(\, value_{new}, read, write\,)\,\Big) \end{aligned} \tag{2.48}$$

Le terme *Property* est paramétré par le nom *value* et les canaux *read* et *write*. Le nom *value* représente la valeur de la propriété et les canaux *read* et *write* permettent respectivement de lire et de modifier cette valeur. De plus, nous définissons un terme *PropertyDirectory* qui modélise un annuaire d'agents *Property*. Cet annuaire est semblable à ceux définis par les termes $TaskDirectory$ (eq. 2.20) et $DHDirectory$ (eq. 2.40).

$$PropertyDirectory(\, add\,) \stackrel{def}{=} !\,\Big(\, add(\, P(\, v, r, w\,)\,)\,.\, P(\, v, r, w\,)\,\Big) \tag{2.49}$$

Le terme *ComputationCase* (eq. 2.50) est une définition d'un cas de calcul. En effet, il regroupe tous les termes qui se rapportent à la définition d'un cas de calcul : les termes $Master$ (eq. 2.44), $TaskDirectory$ (eq. 2.20), $DHDirectory$ (eq. 2.40) et $PropertyDirectory$ (eq. 2.49).

$$\begin{aligned} ComputationCase(\, getTask\,) \stackrel{def}{=} (\,\nu\, add_T, add_{DH}, add_P\,) \\ \left(\begin{array}{ll} & Master(\, getTask, add_T, add_{DH}, add_P\,) \mid TaskDirectory(\, add_T\,) \\ \mid & DHDirectory(\, add_{DH}\,) \mid PropertyDirectory(\, add_P\,) \end{array}\right) \end{aligned} \tag{2.50}$$

Le système (eq. 2.51), permettant le résolution d'un cas de calcul, est maintenant composé de deux termes : l'agent *ComputationCase* (eq. 2.50) et l'agent $Worker^m$ (eq. 2.37). Ces deux termes communiquent à travers le canal $getTask$ qui est commun à tous les agents $Task_i$ (eq. 2.43) du cas de calcul.

$$System \stackrel{def}{=} (\,\nu\, getTask\,)\,\Big(ComputationCase(\, getTask\,) \mid Worker^m(\, getTask\,)\,\Big) \tag{2.51}$$

A ce stade de notre spécification, nous pouvons différencier les termes qui participent à la résolution d'un cas de calcul en deux catégories :

- Les termes dont la définition est indépendante du cas de calcul comme les termes $TaskDirectory$ (eq. 2.20), $DHDirectory$ (eq. 2.40), *PropertyDirectory* (eq. 2.49) et $Worker^m$ (eq. 2.37) ;
- Les termes dont la définition est spécifique à un cas de calcul, comme les termes $Task$ (eq. 2.42) et *ComputeAgent* (eq. 2.41) modélisant une tâche de calcul, le terme *DataHandler* (eq. 2.39) pour les données, le terme *Property* (eq. 2.48) pour les propriétés et le terme $Master$ (eq. 2.44) pour le processus maître.

Le système (eq. 2.51) explicite maintenant clairement la distinction entre le cas de calcul et son évaluation. Autrement dit, il sépare la définition des données et des traitements avec la définition des ressources afin d'assurer la *transparence d'accès* (cf. Section 1.1.2).

2.2.3.5 Modélisation d'un cas de calcul suivant le modèle *SPMD*

Nous décidons de modéliser un système permettant la résolution de l'équation de Laplace par une relaxation de Jacobi, ce cas de calcul est présenté dans la section 2.2.1. Dans le but de simplifier sa modélisation, nous décidons de restreindre le cas de calcul à :

- *2 propriétés « globales »* : le nombre d'itération maximum ($P1$) et la précision atteinte ($P2$). Ces variables sont accessibles en lecture et en écriture par tous les processus participant au cas de calcul.
- *4 blocs de données* : le domaine étudié est une matrice partagée en 4 sous-matrices.
- *4 tâches de calcul* : chaque tâche lit la valeur de la propriété $P1$ et modifie la valeur de la propriété $P2$. De plus, chaque tâche a un paramètre spécifique qui correspond au rang de la sous-matrice traitée.

Dans la suite de cette section, nous définissons les termes permettant de mettre en place ce cas de calcul. Les termes $JacobiTask$ (eq. 2.54), $JacobiMaster$ (eq. 2.55) et $JacobiCA$ (eq. 2.53) qui définissent respectivement une tâche, le processus maître et le code de calcul à exécuter par chaque tâche de ce cas de calcul. Le terme $JacobiCase$ (eq. 2.52) modélise ce cas de calcul comme une substitution appliquéeau terme $ComputationCase$ (eq. 2.50)

$$JacobiCase(\, getTask\,) \stackrel{def}{=} ComputationCase(\, getCase\,)\, \{JacobiMaster/Master\} \tag{2.52}$$

Nous commençons par définir le terme $JacobiCA$ (eq. 2.53) qui modélise le code de calcul exécuté pour une tâche de notre cas de calcul. Ce terme est spécifique à notre cas de calcul et donc différent de celui défini par $ComputeAgent$ (eq. 2.41) : le code qu'il représente lit un bloc de données, lit la valeur de la propriété $P1$ puis modifie les données lues et la valeur de la propriété $P2$.

$$\begin{aligned}
&JacobiCA(\, init, finish\,) \stackrel{def}{=} init(\, ack_{init}\,) \,.\, (\, \nu\, init_{data}, init_{prop}, init_{param}\,) \\
\textcircled{1}\quad &\left(\begin{array}{l} \overline{ack_{init}}\,\langle\, init_{data}, init_{prop}, init_{param}\,\rangle \\ .\ \left(init_{data}(\, input, output\,) \mid init_{prop}(\, read, write\,) \mid init_{param}(\, parameter\,)\right) \end{array} \right) \\
\textcircled{2}\quad &.\left(\begin{array}{l} (\,\nu\, download\,)\left(\overline{input}\,\langle\, download\,\rangle \,.\, download(\, data\,)\right) \\ \mid\ (\,\nu\, getValue\,)\left(\overline{read}\,\langle\, getValue\,\rangle \,.\, getValue(\, property\,)\right) \end{array} \right) \\
\textcircled{3}\quad &.\,\tau\,.\left((\,\nu\, newdata\,)\,\overline{output}\,\langle\, newdata\,\rangle \mid (\,\nu\, newproperty\,)\,\overline{write}\,\langle\, newproperty\,\rangle\right) \\
\textcircled{4}\quad &.\,(\,\nu\, result\,)\,\overline{finish}\,\langle\, result\,\rangle
\end{aligned} \tag{2.53}$$

L'exécution d'un agent mobile défini par le terme $JacobiCA$ (eq. 2.53) peut se décomposer de la façon suivante :

– ① A travers les canaux $init_{data}$, $init_{prop}$ et $init_{param}$, l'agent reçoit les canaux pour accéder aux données entrantes (*input*) et sortantes (*output*), à la propriété à lire (*read*) et à modifier (*write*). L'action $init_{param}(\, parameter \,)$ permet de recevoir le paramètre spécifique à une tâche (*parameter*). Ce dernier est utilisé par l'agent lors de l'exécution du programme représenté par l'action interne τ, c'est pourquoi le nom *parameter* n'est pas visible dans la suite de la définition.
– ② Ensuite, l'agent lit les données avec

$$(\, \nu\, download \,) \left(\overline{input} \langle\, download \,\rangle \;.\; download(\, data \,) \right)$$

et lit la valeur de la propriété avec

$$(\, \nu\, getValue \,) \left(\overline{read} \langle\, getValue \,\rangle \;.\; getValue(\, property \,) \right)$$

– ③ Après avoir exécuté le programme (représenté par l'action interne τ) sur les données (représentées par le nom *data*), l'agent écrit les données modifiées via l'action $\overline{output} \langle\, newdata \,\rangle$ et modifie la valeur de la propriété via l'action $\overline{write} \langle\, newproperty \,\rangle$.
– ④ Enfin, l'agent envoie le résultat à l'agent $Task$ correspondant.

Le terme $JacobiTask$ (eq. 2.54) est défini en fonction du terme $JacobiCA$:
– ① initialisation des données et des propriétés à travers les canaux $init_{data}$ et $init_{prop}$,
– ② demande de la tâche par un agent $Worker$ à travers le canal $getTask$,
– ③ envoi de l'agent mobile $JacobiCA$ à l'agent $Worker$ à travers le canal $worker$ et initialisation de l'agent $JacobiCA$ par les canaux *data*, *prop* et *param* (la notation $(\, \nu\, parameter \,)$ implique que le nom *parameter* est spécifique à un agent $Task$),
– ④ l'agent $Worker$ signale la fin de l'exécution de l'agent mobile CA à travers le canal ack_{finish}.

$$
\begin{aligned}
& JacobiTask(\, init, getTask, finish \,) \stackrel{def}{=} \\
① \; & \left\{ \begin{array}{l} init(\, ack_{init} \,) \;.\; (\, \nu\, init_{data}, init_{prop} \,) \\ \left(\begin{array}{l} \overline{ack_{init}} \langle\, init_{data}, init_{prop} \,\rangle \\ .\;\; \left(init_{data}(\, input, output \,) \;\middle|\; init_{prop}(\, read, write \,) \right) \end{array} \right) \end{array} \right. \\
② \; & .\; getTask(\, worker \,) \;.\; (\, \nu\, init_{agent}, ack_{finish} \,) \\
③ \; & \left\{ \begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} \overline{worker} \langle\, JacobiCA(\, init_{agent}, ack_{finish} \,) \,\rangle \\ .\; (\, \nu\, ack \,) \left(\overline{init_{agent}} \langle\, ack \,\rangle \;.\; ack(\, data, prop, param \,) \right) \end{array} \right) \\ .\; \left(\begin{array}{l} \overline{data} \langle\, input, output \,\rangle \\ \mid\; \overline{prop} \langle\, read, write \,\rangle \\ \mid\; (\, \nu\, parameter \,)\, \overline{param} \langle\, parameter \,\rangle \end{array} \right) \end{array} \right) \\ ④ \; .\; ack_{finish}(\, result \,) \;.\; \overline{finish} \langle\, result \,\rangle \;.\; 0 \end{array} \right.
\end{aligned}
\tag{2.54}
$$

Le processus maître de ce cas de calcul est défini par le terme $JacobiMaster$ (eq. 2.55). A travers le canal add_P, l'agent $JacobiMaster$ ajoute les deux propriétés à l'annuaire défini par le terme $PropertyDirectory$ (eq. 2.49) pour les partager avec les processus travailleurs participant au cas de calcul. La notation $(\,\nu\, v_1, r_1, w_1, v_2, r_2, w_2\,)$ crée les noms et canaux pour paramétrer les deux agents définis par le terme $Property$: l'agent $Property(v_1, r_1, w_1)$ et l'agent $Property(v_2, r_2, w_2)$. Ces deux agents modélisent respectivement la propriété $P1$ et la propriété $P2$. Notons que seul la propriété $P2$ est lue par le processus maître, celle ci est représenté par le nom $prop_2$ reçu par le canal $getValue$. La propriété $P1$ est lue par les agents définis par le terme $JacobiCA$ (eq. 2.53).

$$
\begin{aligned}
&JacobiMaster(\,getTask, add_T, add_{DH}, add_P\,) \stackrel{def}{=} \\
&\quad (\,\nu\, v_1, r_1, w_1, v_2, r_2, w_2\,) \\
&\quad \left(\begin{array}{l} \Big(\overline{add_P}\,\langle\, Property(v_1, r_1, w_1)\,\rangle \;\Big|\; \overline{add_P}\,\langle\, Property(v_2, r_2, w_2)\,\rangle\, \Big) \\ . \;\left(\begin{array}{l} JacobiTaskHandler_1(\,getTask, add_T, add_{DH}, r1, w2\,) \\ \qquad \Big|\; \cdots \;\Big|\; JacobiTaskHandler_n(\,getTask, add_T, add_{DH}, r1, w2\,) \end{array} \right) \end{array} \right) \\
&\quad .\; (\,\nu\, getValue\,) \Big(\overline{r_2}\,\langle\, getValue\,\rangle \;.\; getValue(\,prop_2\,) \Big).\; \tau \;.\; 0
\end{aligned}
\tag{2.55}
$$

L'ordre d'exécution des tâches étant sans importance dans notre exemple, $JacobiMaster$ gère toutes les tâches en parallèle. Les n agents $JacobiTaskHandler_i$ (eq. 2.56) (avec $i = 1, \ldots, n$) modélisent la gestion de ces n tâches.

$$
\begin{aligned}
&JacobiTaskHandler_i(\,getTask, add_T, add_{DH}, r, w\,) \stackrel{def}{=} (\,\nu\, read_i, write_i, init_i, finish_i\,) \\
&\left(\begin{array}{ll}
① & \overline{add_{DH}}\,\langle\, DH_i\,\rangle \;.\; \overline{add_T}\,\langle\, Task_i\,\rangle \\
② & .\; (\,\nu\, ack_{init}\,) \Big(\overline{init_i}\,\langle\, ack_{init}\,\rangle \;.\; ack_{init}(\,data, prop\,) \Big) \\
③ & .\; \Big(\overline{data}\,\langle\, read_i, write_i\,\rangle \;\Big|\; \overline{prop}\,\langle\, r, w\,\rangle \Big) \\
④ & .\; finish_i(\,result_i\,) \\
⑤ & .\; (\,\nu\, download\,) \Big(\overline{read_i}\,\langle\, download\,\rangle \;.\; download(\,data_i\,) \Big)
\end{array} \right)
\end{aligned}
\tag{2.56}
$$

Pour chaque tâche du cas de calcul, modélisée par un agent $JacobiTask_i$, un agent $JacobiTaskHandler_i$ (avec $i = 1, \ldots, n$) réalise les actions suivantes :

- ① mise à disposition de la tâche, modélisé par l'agent $JacobiTask_i$ (eq. 2.54), et du bloc de données, à travers son gestionnaire DH_i (eq. 2.46), sur lequel s'exécute le programme ;
- ② et ③ initialisation de la tâche ;
- ④ récupération du résultat, modélisé par le nom $result_i$, produit lors du traitement de la tâche ;
- ⑤ lecture des données, modélisées par le nom $data_i$, modifiées lors du traitement de la tâche.

Si n est le nombre de transitions nécessaires à l'agent $JacobiCase$ (eq. 2.52) pour que chaque agent $JacobiTaskHandler_i$ ajoute la tâche qu'il traite à $TaskDirectory$ alors

$$JacobiMaster \xrightarrow{\tau}{}^{n} JacobiMaster'$$

qui implique que $JacobiCase'$ est la réduction du terme $JacobiCase$ (eq. 2.52) pour les mêmes transitions.

$$JacobiCase'(\, getTask \,) \stackrel{def}{=} (\, \nu\, add_T, add_{DH}, add_P \,) \left(\begin{array}{l} \quad \boldsymbol{JacobiMaster'(\, getTask, add_T, add_{DH}, add_P \,)} \\ \mid \quad \boldsymbol{Task_1} \mid \boldsymbol{Task_2} \mid \boldsymbol{Task_3} \mid \boldsymbol{Task_4} \mid TaskDirectory(\, add_T \,) \\ \mid \quad \boldsymbol{DH_1} \mid \boldsymbol{DH_2} \mid \boldsymbol{DH_3} \mid \boldsymbol{DH_4} \mid DHDirectory(\, add_{DH} \,) \\ \mid \quad \boldsymbol{Property_1} \mid \boldsymbol{Property_2} \mid PropertyDirectory(\, add_P \,) \end{array} \right) \tag{2.57}$$

où $DH_i \stackrel{def}{=} DataHandler(read_i, write_i)$ avec $i = 1, \ldots, 4$,

$Task_i \stackrel{def}{=} JacobiTask(init_i, getTask, finish_i)$ avec $i = 1, \ldots, 4$

$Property_1 \stackrel{def}{=} Property(v_1, r_1, w_1)$ et $Property_2 \stackrel{def}{=} Property(v_2, r_2, w_2)$

Enfin, lorsque toutes les tâches ont été traitées et les données modifiées ($data_i$) récupérées, l'agent $JacobiMaster$ (eq. 2.55) lit la valeur de la propriété modélisée par l'agent $Property_2$ et effectue un traitement final (modélisé par l'action τ).

2.2.3.6 Bilan

Le terme $ComputationCase$ (eq. 2.50) modélise un cas de calcul. Cet agent communique avec la ferme de travailleurs afin de résoudre le cas de calcul qu'il modélise. L'exécution d'un agent $ComputationCase$ ne demande pas de connaître le nombre d'agents $Worker$ qui constituent la ferme. Cette caractéristique permet d'avoir une certaine *transparence de concurrence* (cf. Section 1.1.2). L'exemple simplifié de la relaxation de Jacobi, nous a permis de présenter les réductions successives d'un terme modélisant un cas de calcul. Cette validation fonctionnelle n'est qu'une étape, la section suivante s'intéresse à l'impact d'un cas de calcul sur un autre et plus particulièrement la disponibilité des ressources de calcul.

2.2.4 Modélisation d'un « espace de travail »

A ce stade de notre spécification, le système proposé par l'équation 2.51 définit qu'une ferme de travailleurs est liée à un seul cas de calcul. Afin de permettre aux différents processus travailleurs

qui composent cette « ferme » de participer à plusieurs cas de calcul, nous proposons, dans cette section, de faire évoluer les termes $ComputationCase$ et $Worker$.

2.2.4.1 L'espace de travail

Le terme $ComputationCase$, défini par l'équation 2.50, évolue pour permettre la gestion des processus travailleurs participant au cas de calcul qu'il modélise. Aux trois annuaires déjà présents, définis par les termes $TaskDirectory$ (eq. 2.20), $DHDirectory$ (eq. 2.40) et $PropertyDirectory$ (eq. 2.49), le terme $ComputationCase$ (eq. 2.58) contient un gestionnaire du cas de calcul, représenté par le terme $CaseHandler$ (eq. 2.59).

$$
\begin{aligned}
&ComputationCase(\,\boldsymbol{init}\,) \stackrel{def}{=} \boldsymbol{init(\,getCase\,)} \\
&\quad .\,(\,\nu\, add_T, add_{DH}, add_P, getTask, \boldsymbol{finish}\,) \\
&\quad \left(\begin{array}{l}
\quad Master(\,getTask, add_T, add_{DH}, add_P, \boldsymbol{finish}\,) \\
\mid\; TaskDirectory(\,add_T\,) \mid DHDirectory(\,add_{DH}\,) \mid PropertyDirectory(\,add_P\,) \\
\mid\; \boldsymbol{CaseHandler(\,getCase, getTask, finish\,)}
\end{array}\right)
\end{aligned}
\tag{2.58}
$$

Ce gestionnaire assure le cycle de vie du cas de calcul et des processus travailleurs participant à sa résolution. Pour participer au cas de calcul, un processus travailleur envoie à travers le canal $getCase$ un canal $worker$. A travers ce canal, le gestionnaire envoie à l'agent $Worker$ le canal $getTask$ commun à toutes les tâches de ce cas de calcul. Notons que le gestionnaire $CaseHandler$ reçoit aussi un canal $stop$ à travers lequel il préviendra ce processus travailleur de la fin du cas de calcul. Nous détaillons plus précisément l'utilisation de ce canal $stop$ dans la section 2.2.4.2.

$$
\begin{aligned}
&CaseHandler(\,getCase, getTask, finish\,) \stackrel{def}{=} \\
&\qquad \left(\begin{array}{l}
\quad getCase(\,worker, stop\,)\;.\;\overline{worker}\,\langle\, getTask\,\rangle \\
.\quad (\,\nu\, next\,) \left(\begin{array}{l}
\quad \Big(finish.\;\overline{stop}\;.\;\overline{next}\Big) \\
\mid\; CaseHandler(\,getCase, getTask, next\,)
\end{array}\right)
\end{array}\right) \\
&\quad + \quad \Big(finish\;.\;0\Big)
\end{aligned}
\tag{2.59}
$$

Le terme $Master$ (eq. 2.44) évolue pour permettre à un processus maître de prévenir le gestionnaire, modélisé par le terme $CaseHandler$, de la fin du cas de calcul. En effet, le terme $Master$ (eq. 2.60) et $CaseHandler$ (eq. 2.59) communiquent à travers le canal $finish$ (eq. 2.58).

$$Master(\, getTask, add_T, add_{DH}, add_P, \boldsymbol{finish}\,) \stackrel{def}{=} \left(\begin{array}{c} TaskHandler_1(\, getTask, add_T, add_{DH}\,) \\ \quad \mid \cdots \mid TaskHandler_n(\, getTask, add_T, add_{DH}\,) \end{array} \right) . \tau . \overline{\boldsymbol{finish}} \tag{2.60}$$

L'espace de travail doit être capable de contenir différents cas de calcul simultanément. Ces cas de calcul doivent être accessibles par les processus travailleurs disponibles. Le terme $CaseDirectory$ (eq. 2.61) modélise un annuaire de cas de calcul qui peut être considéré comme notre espace de travail.

$$CaseDirectory(\, add, getCase\,) \stackrel{def}{=} !\left(add(\, CC(i)\,) \,.\, \Big(CC(\, i\,) \mid \overline{i}\,\langle\, getCase\,\rangle\, \Big) \right) \tag{2.61}$$

Pour modéliser l'ajout d'un cas de calcul à l'« espace de travail », nous définissons un terme $User$ (eq. 2.62) représente un utilisateur de l'espace de travail qui ajoute un cas de calcul.

$$User(\, add_{CC}\,) \stackrel{def}{=} (\,\nu\, init\,)\, \overline{add_{CC}}\,\langle\, ComputationCase(init)\,\rangle \tag{2.62}$$

En ajoutant un agent $User$ et un agent $CaseDirectory$ à la ferme de travailleurs $Worker^m$, nous obtenons le terme $System$ défini par l'équation 2.63.

$$System \stackrel{def}{=} (\,\nu\, add_C, getCase\,) \Big(CaseDirectory(\, add_C, getCase\,) \mid Worker^m(\, getCase\,) \mid User(\, add_C\,) \Big) \tag{2.63}$$

Les processus $CaseDirectory$ et $User$ partagent le nom add_C ce qui permet au processus $User$ d'envoyer le terme d'ordre supérieur $ComputationCase$ au processus $CaseDirectory$. Cette transition modélise l'ajout d'un cas de calcul à l'espace de travail. Le gestionnaire du cas de calcul est alors en attente sur le canal $getCase$ (eq. 2.59) et peut communiquer avec les processus travailleurs qui composent la ferme de travailleurs, modélisée par le terme $Worker^m$.

2.2.4.2 Des processus travailleurs « adaptables »

Le terme $Worker$ (eq. 2.28) définit un processus travailleur capable de traiter toutes les tâches d'un cas de calcul. En effet, l'utilisation d'un agent mobile, modélisé par le terme $ComputeAgent$ (eq. 2.41), rend le processus travailleur spécifique à une tâche après la réception d'un agent mobile à travers le canal $task$. Cette modélisation impose au processus travailleur de participer à un seul cas de calcul. Nous faisons évoluer notre modélisation d'un processus travailleur afin de permettre à celui-ci de pouvoir participer à plusieurs cas de calcul lors de son exécution. Le but

étant que le processus travailleur puisse participer à tous les cas de calcul présents dans l'espace de travail.

$$Worker(\,getCase\,) \stackrel{def}{=} Worker_{started}(\,getCase\,) \tag{2.64}$$

Un processus travailleur, modélisé par le terme $Worker$ (eq. 2.64), peut se trouver dans deux états. Ces états sont modélisés par deux termes différents :

– Le terme $Worker_{started}$ qui modélise un processus travailleur ne participant à aucun cas de calcul. Cet état correspondant à l'état initial d'un processus travailleur.

$$\begin{aligned} Worker_{started}(\,getCase\,) \stackrel{def}{=}\ & (\,\nu\, space, stop\,)\,\Big(\overline{getCase}\,\langle\, space, stop\,\rangle\ .\ space(\,getTask\,)\,\Big) \\ & .\ Worker_{connected}(\,getTask, stop\,)\ .\ Worker_{started}(\,getCase\,) \end{aligned} \tag{2.65}$$

  A travers les communications réalisées par les canaux $getCase$ et $space$, l'agent $Worker_{started}$ récupère le canal $getTask$ commun à tous les agents définis par le terme $Task$ d'un cas de calcul.

– Le terme $Worker_{connected}$ qui modélise un processus travailleur participant à un cas de calcul.

$$\begin{aligned} Worker_{connected}(\,getTask, stop\,) \stackrel{def}{=}\ & \left(\begin{array}{l} (\,\nu\, task\,)\,\Big(\overline{getTask}\,\langle\, task\,\rangle\ .\ task(\,CA(\,init, finish\,)\ )\Big) \\ .\quad CA(\,init, finish\,)\ .\ Worker_{connected}(\,getTask, stop\,) \end{array} \right) \\ +\quad & \Big(stop\ .\ 0\Big) \end{aligned} \tag{2.66}$$

  Ce terme est très proche du terme $Worker$ défini précédemment (eq. 2.28). Néanmoins, une différence est à noter : le canal *stop*. Ce canal permet à l'agent $ComputationCase$ associé au cas de calcul de signaler la fin de son exécution à l'agent $Worker_{connected}$.

Pour participer à la résolution d'un cas de calcul, un processus travailleur, modélisé par le terme $Worker$ (eq. 2.64), communique avec le gestionnaire de ce cas de calcul (eq. 2.67) pour récupérer le canal $getTask$ et lui envoyer un canal *stop*.

$$\begin{array}{lcl} & CaseHandler(\,getCase, getTask, finish\,) & \Big|\ Worker(\,getCase\,) \\ \xrightarrow{\tau}\xrightarrow{\tau} & CaseHandler' & \Big|\quad \cdots .\ \boldsymbol{stop} \end{array} \tag{2.67}$$

Le terme $CaseHandler'$ (eq. 2.68) est la réduction du terme $CaseHandler$ après les deux réductions successives qui modélisent l'ajout d'un processus travailleur dans la résolution du cas de calcul.

$$CaseHandler' \equiv \Big(finish.\ \overline{\boldsymbol{stop}}\ .\ \overline{next}\ \Big)\ \Big|\ CaseHandler(\,getCase, getTask, next\,) \tag{2.68}$$

La participation d'un nouveau processus travailleur au calcul entraine la réduction du terme $CaseHandler'$ en $CaseHandler''$ (eq. 2.69).

$$CaseHandler'' \equiv \Big(finish.\ \overline{stop}\ .\ \overline{next} \Big) \mid \Big(next.\ \overline{stop_2}\ .\ \overline{next_2} \Big) \mid \underbrace{\Big(\cdots + next_2\ .\ 0 \Big)}_{CaseHandler(\, getCase,\ getTask,\ next_2\,)} \tag{2.69}$$

Le canal $finish$ du terme $CaseHandler''$ est le canal créé dans le terme $ComputationCase$ (eq. 2.58). Ce canal est commun avec l'agent défini par le terme $Master$ et à travers ce canal, l'agent $Master$ (eq. 2.60) communique la fin du cas de calcul au gestionnaire défini par le terme $CaseHandler$. À la réception sur le canal $finish$, le gestionnaire informe les agents $Worker$ (deux dans notre exemple (eq. 2.69)) participant au cas de calcul de la fin de celui-ci à travers les canaux $stop$ et $stop_2$.

Les deux processus travailleurs qui participaient au cas de calcul sont de nouveau disponibles pour participer à un autre cas de calcul. Ils sont évalués comme le terme $Worker_{started}$ (eq. 2.65).

2.3 Conclusion

L'utilisation du π-calcul polyadique d'ordre supérieur nous a permis de définir formellement une architecture logicielle pour le calcul numérique. Un système distribué bâti sur cette architecture est capable de résoudre différents cas de calcul numérique. Nous avons fait évoluer notre modélisation en partant du modèle classique *Maître-Travailleurs* (cf. Section 1.1.3) pour obtenir au final la modélisation d'une architecture adaptable (cf. Section 2.2.4), c'est à dire une architecture capable de s'adapter aussi bien au nombre de ressources disponibles qu'au nombre de cas de calcul qu'elle doit traiter.

La spécification $MCA\pi Spec$, résultat de la modélisation en π-calcul de cette architecture, définit un système (eq. 2.70) composé des termes $CaseDirectory$ (eq. 2.61), modélisant un espace de travail, et $Worker^m$, modélisant une ferme de m travailleurs définis par le terme $Worker$ (eq. 2.64). Il est possible d'ajouter un agent $User$ (eq. 2.62) à ce système pour démarrer la résolution d'un cas de calcul.

$$System \stackrel{def}{=} (\,\nu\, add_C, getCase\,) \Big(CaseDirectory(\, add_C, getCase\,) \mid Worker^m(\, getCase\,) \Big) \tag{2.70}$$

Nos spécification nous ont permis d'exprimer nos exigences présentées dans le chapitre 1 comme la transparence de localisation avec les $ComputeAgents$, la transparence d'échelle avec ou encore la transparence d'accès avec la distinction entre un cas de calcul et son évaluation.

Afin de vérifier les propriétés de notre architecture, nous allons, dans le chapitre suivant, transformer celle-ci en un réseaux d'automates temporisés. Nous établirons alors ces propriétés temporelles par model-checking.

Chapitre 3

Model-checking appliqué à un système d'agents mobiles

Sommaire

L'objectif de ce chapitre est de vérifier des propriétés de la spécification π-calcul écrite dans le chapitre précédent. Pour cela, nous utilisons la logique TCTL. Nous présentons dans la section 3.1 les techniques de vérifications des systèmes temps-réel qui nous ont conduit à choisir l'outil UppAal pour vérifier notre système. La section 3.2 définit la transformation d'un système défini dans un algèbre de processus (en l'occurrence en π-calcul polyadique d'ordre supérieur présenté et utilisé dans le chapitre 2) en un système d'automates. Enfin, dans la section 3.3, nous transformons notre système défini dans le chapitre 2 en un réseau d'automates temporisés afin de pouvoir vérifier certaines propriétés temporelles [Mat98] grâce à l'outil UppAal.

3.1 Vérification des systèmes temps-réel

Dans la littérature des algèbres de processus, il existe plusieurs approches de la vérification. L'une d'elles consiste à prendre deux descriptions d'un même système et de prouver leur équivalence sous certaines conditions. Généralement, une description est appelée la spécification et l'autre l'implémentation qui est souvent écrite en respectant des détails techniques supplémentaires. Cette preuve d'équivalence utilise une propriété souvent structurelle sur la spécification. Cette forme de vérification a ses limites, en particulier dans le cas de spécification partielle. Cette approche s'intéresse à certains aspects du système et ignore totalement les autres.

Une autre approche, plus adaptée dans notre cas, est basée sur l'emploi de logique modale ou temporelle. Ces logiques ne possèdent pas les mêmes structures sous-jacentes que les algèbres de processus, aussi, elles permettent d'écrire des spécifications ayant une autre forme d'abstraction. A partir de telles spécifications, il est alors possible d'évaluer une formule logique et de conclure si elle respecte ou pas le modèle exprimé en utilisant une algèbre de processus.

Les logiques utilisées ont souvent un faible pouvoir d'expression et il est même parfois difficile d'exprimer qu'un comportement est cyclique ou non. Des logiques plus intéressantes telles que le modal μ-calcul [AN01], associé à une algèbre comme le π-calcul, ont une définition basée sur des systèmes de transitions étiquetées. Cela rend la construction de tels modèles plus facile à obtenir.

D'autres logiques temporelles nous permettent d'utiliser des outils puissants tels que des Model-Checkers. Ainsi, un modèle à états finis est construit depuis une spécification afin de faire de la preuve. Il existe plusieurs formes de logiques temporelles avec des opérateurs pour exprimer des propriétés du type : « *pour toutes les futures exécutions, il y a ...* »

3.1.1 Systèmes d'automates

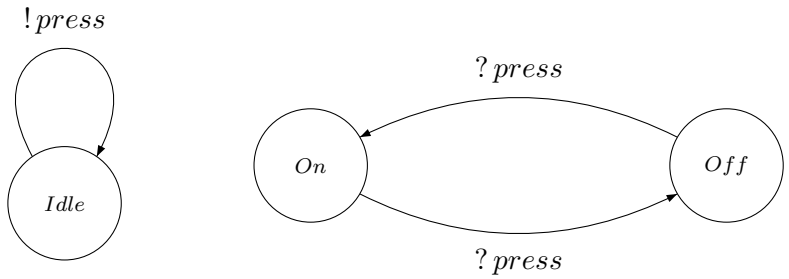
Les systèmes qui se prêtent le mieux aux techniques de model-checking sont ceux facilement représentables par des automates (cf Définition 3.1). Ainsi, toute description à base de comportement à états tel que le processus $Worker$ spécifié dans le chapitre 2 est un bon candidat à une telle transformation.

Définition 3.1 (Automates) *Un automate A est défini par le quintuplet $\langle Q, E, T, q_0, \ell \rangle$ où :*
- *Q est un ensemble fini d'états ;*
- *E est l'ensemble fini des étiquettes des transitions ;*
- *$T \subseteq Q \times E \times Q$ est l'ensemble des transitions ;*
- *q_0 est l'état initial de l'automate*
- *ℓ est l'application qui associe à tout état de Q l'ensemble fini des propriétés élémentaires vérifiées dans cet état.*

La modélisation de systèmes réels nécessite la plupart du temps la manipulation de *variables d'états*. Ces variables sont le plus souvent utilisées en tant que compteur, comme par exemple pour compter un nombre d'erreur ou de passage par une transition.

Lorsque l'on s'intéresse à un système complexe, il est souvent plus simple de le découper en sous-systèmes (ou modules). De la même manière, pour construire la modélisation globale d'un système de ce type, il est nécessaire de modéliser chaque sous-système. L'automate global modélisant le système est ainsi obtenu en synchronisant les automates de chaque module. Il existe de nombreuses manières de réaliser cette *synchronisation* mais le résultat, appelé le *produit synchronisé*, entraine une explosion du nombre d'états, et la modélisation de l'automate global en devient quasiment impossible.

Le produit synchronisé de ces automates est un automate dont l'espace d'états est le produit des espaces d'états des automates composants, l'état initial est le n-uplet des états initiaux, l'alphabet d'actions est l'union des alphabets, et une transition Le produit synchronisé offert par la méthode de synchronisation par message est un cas particulier. Il s'agit de faire communiquer les différents automates modélisant le système global par l'envoi/réception de messages. L'émission d'un message m est notée $!m$ alors que la réception correspondante est notée $?m$. Pour que l'automate soit valide, chaque émission doit correspondre à une réception. La figure 3.1 donne un exemple de ce type de réseau avec la modélisation d'une lampe et d'un interrupteur.



(a) Interrupteur (b) Lampe

FIGURE 3.1 – Exemple d'un réseau d'automates : la modélisation d'une lampe.

3.1.2 La représentation du temps

Il n'est pas réaliste de considérer que la durée d'un traitement est fixe, le traitement pouvant se terminer plus tôt ou plus tard que prévu, sans compter les pannes possibles de machines ou encore les pertes de messages provoquant un ralentissement global du système.

Deux méthodes de modélisation du temps s'opposent : le *temps discret* et le *temps dense*. Notons que dans cette thèse, nous considérons que le temps s'écoule de façon continue. La notion de

temps dense s'oppose à la notion de temps discret dans lequel un grain minimal d'écoulement du temps est défini, c'est à dire que rien ne peut se passer dans une période de temps plus petite. Ce point de vue, bien que souvent très proche de la réalité et bien adapté à de nombreux cas, semble une hypothèse forte pour des systèmes distants ne partageant pas d'horloges au sens synchrone. Il existe cependant des travaux considérant une modélisation discrète du temps [Mag07].

La sémantique des modèles temporisés s'exprime en termes de systèmes de transitions temporisés (STT) où le domaine de temps, que nous notons $\mathbb{T}$, peut être l'ensemble $\mathbb{N}$ des entiers naturels, l'ensemble $\mathbb{Q}_{\geq 0}$ des rationnels positifs ou nuls, ou l'ensemble $\mathbb{R}_{\geq 0}$ des réels positifs ou nuls. Nous supposons par la suite que le domaine de temps $\mathbb{T}$ est l'ensemble $\mathbb{R}_{\geq 0}$. Nous donnons la définition formelle :

Définition 3.2 (Système de transitions temporisé) *Un système de transitions temporisé (STT) est un quadruplet* $\mathcal{T} = (S, s_0, \rightarrow, \Sigma)$ *où :*
- S *est un ensemble d'états de contrôle ;*
- s_0 *est l'état de contrôle initial ;*
- $\rightarrow \subseteq S \times (\mathbb{T} \cup \Sigma) \times S$ *est la relation de transition ;*
- Σ *est un ensemble d'actions.*

Deux types de transitions sont possibles pour ces systèmes de transitions temporisés :
- Les transitions $\xrightarrow{a}$, avec $a \in \Sigma$, qui correspondent à des actions au sens usuel et considérées comme instantanées ;
- les transitions $\xrightarrow{d}$, avec $d \in \mathbb{T}$, qui expriment l'écoulement d'une durée d et vérifient les conditions particulières suivantes :
  - *délai nul :* $s \xrightarrow{0} s'$ si et seulement si $s' = s$;
  - *additivité :* si $s \xrightarrow{d} s'$ et $s' \xrightarrow{d'} s''$, alors $s \xrightarrow{d+d'} s''$;
  - *déterminisme temporel :* si $s \xrightarrow{d} s_1$ et $s \xrightarrow{d} s_2$, alors $s_1 = s_2$;
  - *continuité :* si $s \xrightarrow{d} s'$, alors pour tout d' et d'' tels que $d = d' + d''$, il existe s'' tel que $s \xrightarrow{d'} s'' \xrightarrow{d''} s'$.

L'exécution d'un STT est une séquence finie ou infinie de transitions continues et discrètes de S. On peut écrire une exécution ρ d'un STT sous la forme suivante :

$$\rho = q_0 \xrightarrow{d_0} q'_0 \xrightarrow{a_0} q_1 \xrightarrow{d_1} q'_1 \xrightarrow{a_1} \cdots q_n \xrightarrow{d_n} q'_n \cdots$$

3.1.2.1 Les automates temporisés

Les automates temporisés (proposés par Alur et Dill en 1994 [AD94]) étendent les automates finis classiques avec la notion de temps. Ce dernier est alors ajouté au modèle classique sous la forme d'horloges qui évoluent de manière continue avec le temps. Des prédicats peuvent être appliqués sur ces horloges. Ces prédicats sont de deux types :

– les *gardes* donnent des contraintes sur les horloges à respecter pour pouvoir exécuter une transition d'action ;
– les *invariants* donnent des contraintes à respecter pour rester dans un état.

On note $\mathcal{C}(X)$ l'ensemble des contraintes d'horloges sur X, c'est à dire l'ensemble des combinaisons booléennes de contraintes atomiques de la forme $x \sim c$ où x est une horloge ($x \in X$), c une constante ($c \in \mathbb{N}$) et $\sim$ un opérateur de comparaison ($\sim \in \{=, <, \leq, >, \geq\}$)

Formellement, un automate temporisé est défini comme suit :

Définition 3.3 (Automate temporisé) *Un automate temporisé $\mathcal{A}$ est un 6-uplet $(Q, X, q_0, T, Inv, \Sigma)$ où :*
– *Q est un ensemble fini d'états de contrôle ou localités ;*
– *X est un ensemble fini d'horloges ;*
– *q_0 est la localité initiale de l'automate ;*
– *$T \subseteq Q \times \mathcal{C}(X) \times \Sigma \times 2^X \times Q$ est un ensemble fini de transitions ;*
– *$Inv : Q \to \mathcal{C}(X)$ associe un invariant à chaque localité ;*
– *Σ est un alphabet d'action.*

La transition e, avec $e = \langle q, g, a, r, q' \rangle \in T$, exprime un passage possible de q à q' avec g la garde associée, a l'étiquette de e et r l'ensemble des horloges devant être remises à zéro. On note aussi cette transition $q \xrightarrow{g,a,r} q'$.

Les contraintes d'horloges (gardes et invariants) sont interprétées sur des valuations d'horloges. Une valuation v pour X est une fonction ($v : X \to \mathbb{R}_{\geq 0}$) qui associe à chaque horloge x sa valeur $v(x)$. On note $\mathbb{R}^X_{\geq 0}$ l'ensemble des valuations pour X. Chaque état d'un automate temporisé est alors une paire $(q, v) \in Q \times \mathbb{R}^X_{\geq 0}$ où $q \in Q$ et v est une valuation d'horloges satisfaisant l'invariant de la localité q.

La figure 3.2 présente un exemple d'automate temporisé modélisant une lampe qui s'allume faiblement (état *Low*) lorsque l'interrupteur (cf. Figure 3.1a) est appuyé (action modélisée par une transition étiquetée par *!press*). Cette action remet l'horloge h à zéro. Si l'interrupteur est appuyé une seconde dans les 5 secondes qui suivent ($h \leq 5$), la lampe s'allume plus fortement (état *Bright*), sinon la lampe s'éteint.

Les valeurs des horloges sont modifiées de deux façons :
– soit lors d'une *transition continue* (ou *transition de temps*). Si un certain délai $d \in \mathbb{T}$ s'écoule, alors les valeurs de toutes les horloges s'incrémentent de d. On note $v + d$ la valuation qui associe à l'horloge x la valeur $v(x) + d$. L'automate passe alors de l'état (q, v) à l'état $(q, v + d)$.
– soit lors d'une *transition discrète* (ou *transition d'action*). Dans ce cas, les mises à jour des horloges sont limitées à des remises à zéro. Pour $r \subseteq X$, $[r \leftarrow 0]v$ représente la valuation v' définie par : $v'(x) = 0$ pour tout $x \in r$ et $v'(x) = v(x)$ pour $x \in X \setminus r$.

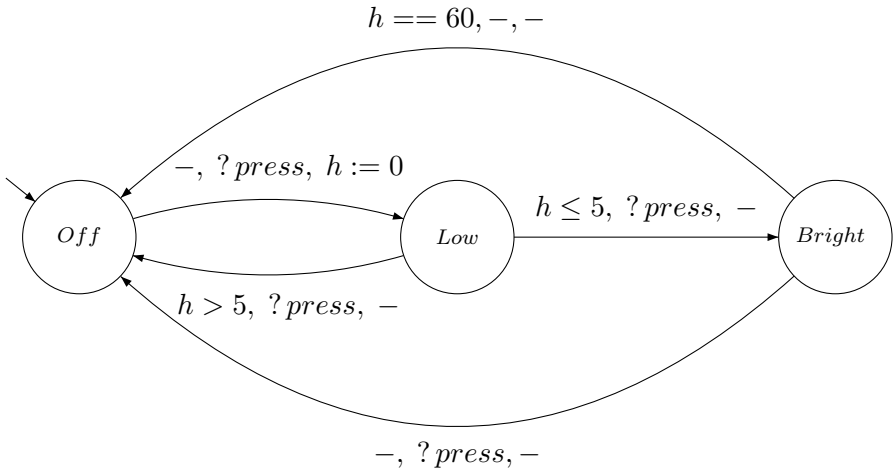


FIGURE 3.2 – Exemple d'un automate temporisé

Définition 3.4 (Sémantique des automates temporisés) *La sémantique d'un automate* $\mathcal{A} = (Q, X, q_0, T, Inv, \Sigma)$ *est définie par le STT* $\mathcal{T}_A = (S, s_0, \rightarrow, \Sigma)$ *où :*
- $S = \{(q, v) \in Q \times \mathbb{R}^X_{\geq 0} \mid v \models Inv(q)\}$;
- $s_0 = (q_0, v_0)$ *avec* $v_0(x) = 0$ *, pour tout* $x \in X$;
- *la relation de transition* $\rightarrow$ *correspond à deux types de transitions :*
  - *les transitions d'actions :* $(q, v) \xrightarrow{a} (q', v')$ *si et seulement s'il existe* $q \xrightarrow{g,a,r} q' \in T$ *tel que* $v \models g,\ v' = [r \leftarrow 0]v$ *et* $v' \models Inv(q')$.
  - *les transitions de temps : si* $d \in \mathbb{R}_{\geq 0},\ (q, v) \xrightarrow{d} (q, v+d)$ *si et seulement si* $v+d \models Inv(q)$.

Comme tout système de transitions temporisé, l'exécution d'un automate temporisé commence par sa configuration initiale : à partir de son état initial q_0 avec toutes les horloges à zéro. Ensuite, il effectue successivement des transitions qui peuvent être de deux types :
- les transitions d'actions (si la valeur des horloges le permet) qui remettent à zéro certaines horloges ;
- les transitions de temps qui incrémentent toutes les horloges d'une même durée en respectant l'invariant associé à la localité courante.

Par exemple, l'automate temporisé $\mathcal{A}_1$ de la figure 3.2 peut évoluer à partir de sa configuration initiale $(idle, 0)$ de la façon suivante :

$$(Off, 0) \longrightarrow (Off, 10.7) \xrightarrow{?press} (Low, 0) \longrightarrow (Low, 3.7) \xrightarrow{?press} (Bright, 3, 7) \longrightarrow (Bright, 16.1) \xrightarrow{?press} (Off, 16.1) \cdots$$

Dans les réseaux d'automates temporisés, tous les automates s'exécutent en parallèle et à la même vitesse. Leurs horloges sont toutes synchronisées sur le même temps global et le partage d'horloges entre plusieurs automates du réseau est tout à fait autorisé. On utilise la notation $(\vec{q}, v)$ pour désigner la configuration d'un réseau où $\vec{q}$ est un vecteur de localités et v une fonction associant à

chaque horloge du réseau sa valeur à l'instant courant. Le comportement d'un système complexe peut être représenté par un unique automate temporisé qui résulte du produit synchronisé de plusieurs autres (cf. Définition 3.5).

Définition 3.5 (Produit synchronisé) *Soient* $\mathcal{A}_1 = (Q_1, X_1, q_0^1, T_1, Inv_1, \Sigma_1)$ *et* $\mathcal{A}_2 = (Q_2, X_2, q_0^2, T_2, Inv_2, \Sigma_2)$ *deux automates temporisés avec* $X_1 \cap X_2 = \emptyset$ *alors la synchronisation de* $\mathcal{A}_1$ *et* $\mathcal{A}_2$ *est l'automate temporisé* $\mathcal{A}_1 k \mathcal{A}_2 = (Q, X, q_0, T, Inv, \Sigma)$ *où*

- $Q = Q_1 \times Q_2$
- $X = X_1 \cup X_2$
- $q_0 = (q_0^1, q_0^2)$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- *Si* $\langle q_1, g_1, a_1, r_1, q_1' \rangle \in T_1$ *et* $\langle q_2, g_2, a_2, r_2, q_2' \rangle \in T_2$ *alors* T *est défini par :*
  - *si* $a_1 = a_2 = a \in \Sigma_1 \cap \Sigma_2$ *alors* $\langle (q_1, q_2), g_1 \wedge g_2, a, r_1 \cup r_2, (q_1', q_2') \rangle \in T$
  - *si* $a_1 \in \Sigma_1 \setminus \Sigma_2$ *alors* $\langle (q_1, q_2), g_1, a_1, r_1, (q_1', q_2) \rangle \in T$
  - *si* $a_2 \in \Sigma_2 \setminus \Sigma_1$ *alors* $\langle (q_1, q_2), g_2, a_2, r_2, (q_1, q_2') \rangle \in T$
- $\forall (q_1, q_2) \in Q_1 \times Q_2, Inv(q_1, q_2) = Inv_1(q_1) \wedge Inv_2(q_2)$

3.1.3 Logique temporelle

Une logique temporelle sert à énoncer des propriétés portant sur les exécutions d'un système. Ces propriétés font intervenir la notion d'ordonnancement dans le temps, comme par exemple : « une lampe s'allume après avoir appuyer sur l'interrupteur ».

3.1.3.1 La logique temporelle CTL

La logique CTL (pour Computation Tree Logic) a été définie au début des années 1980 dans [EH86]. Elle est interprétée sur des structures de Kripke, c'est à dire des automates finis dont les états sont étiquetés par des propositions atomiques. Elle permet d'exprimer des propriétés sur ces états.

Définition 3.6 (Syntaxe de *CTL*) *Les formules de CTL sont décrites par la grammaire suivante :*

$$\begin{aligned} \varphi, \psi ::= \; & P_1 \mid P_2 \mid \cdots \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid \\ & EX\varphi \mid EF\varphi \mid EG\varphi \mid E\varphi U\psi \mid AX\varphi \mid AF\varphi \mid AG\varphi \mid A\varphi U\psi \end{aligned}$$

où P_i *sont des propositions atomiques*[6] *qui parlent des états. Elles donnent pour un état donné une valeur de vérité bien définie.*

6. Les propositions atomiques (ou élémentaires) sont des phrases simples dont on peut déterminer dans un contexte (ou interprétation) si elles sont vraies ou fausses.

En CTL, chaque occurrence d'un combinateur temporel (**U**, **X**, **F** ou **G**) doit être immédiatement sous la portée d'un quantificateur de chemin (**A** ou **E**).

Ainsi, par exemple, $\mathbf{E}(\varphi \mathbf{U} \psi)$ signifie qu'il existe un chemin partant de l'état courant, et tel que ψ sera vraie dans un état futur, et que tous les états intermédiaires vérifieront φ. De son coté, $\mathbf{AX}\varphi$ signifie que le long de tout chemin partant de l'état courant, le successeur de l'état courant satisfait φ. En d'autres termes, cela signifie que tous les successeurs de l'état courant satisfont φ.

$\mathbf{EF}\varphi$ signifie qu'il existe un chemin le long duquel φ est vérifiée à une certaine position et donc qu'il est possible d'arriver à un état vérifiant φ. $\mathbf{AF}\varphi$ signifie que φ est vraie à une certaine position le long de tout chemin, c'est à dire que φ est inévitable. $\mathbf{AG}\varphi$ indique que φ est toujours vraie pour tout état accessible. Enfin, $\mathbf{EG}\varphi$ signifie qu'il existe un chemin le long duquel φ est toujours vraie.

Les combinateurs booléens permettent généralement de relier plusieurs sous-formules grâce à la négation $\neg$, à la conjonction $\wedge$ (« et »), à la disjonction $\vee$ (« ou ») et à l'implication logique $\Rightarrow$.

3.1.3.2 Temporisation de CTL

La description d'un système sous la forme d'un réseau d'automates temporisés permet d'énoncer des propriétés avec la logique CTL sur ce système. Ces propriétés sont alors uniquement temporelles et ne mettent pas en jeu les informations quantitatives fournies par les horloges. La possibilité de pouvoir énoncer des propriétés *temps-réel* devient nécessaire. Pour cela, nous utilisons une logique temporisée qui est une extension d'une logique temporelle par des primitives permettant d'exprimer des conditions sur les durées et les dates. La logique TCTL [HNS+94], version temporisée de CTL, fournit ce langage logique pour spécifier des propriétés temporisées adaptées aux systèmes temps-réels.

Définition 3.7 (Syntaxe de *TCTL*) *Les formules de TCTL sont décrites par la grammaire suivante :*

$$\begin{aligned} \varphi, \psi ::= \; & P_1 \mid P_2 \mid \cdots \\ & \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \\ & \mid EF_{(\sim k)}\varphi \mid EG_{(\sim k)}\varphi \mid E\varphi U_{(\sim k)}\psi \\ & \mid AF_{(\sim k)}\varphi \mid AG_{(\sim k)}\varphi \mid A\varphi U_{(\sim k)}\psi \end{aligned}$$

avec $c \in \mathbb{N}$ *et* $\sim \in \{=, <, >, \leq, \geq\}$.

Il existe différentes familles de propriétés [Sch00] :

- **Les propriétés d'atteignabilité.** Ce type de propriété permet d'énoncer qu'une certaine situation peut être atteinte. Généralement il est surtout intéressant d'utiliser la négation d'une telle propriété. Pour exprimer ce type de propriété en logique temporelle on utilise le combinateur EF en écrivant $EF\phi$, ce qui signifie « il existe un chemin partant de l'état courant et sur lequel se trouve un état vérifiant ϕ ».
- **Les propriétés de sûreté.** Dans ce type de propriété, on cherche à énoncer que, sous certaines conditions, quelque chose ne se produira jamais. Dans des cas simples, ces propriétés peuvent être vues comme la négation des propriétés d'atteignabilité. Le combinateur AG exprime ce type de propriétés en logique temporelle.
- **Les propriétés de vivacité.** Par ce type de propriétés on énonce que, sous certaines conditions, quelque chose finira par avoir lieu. Ces propriétés sont plus « exigeantes » que les propriétés d'atteignabilité. Ces propriétés paraissent peu utiles car elles n'apportent aucune information : elles sont bien trop abstraites. Les systèmes temporisés apportent la notion de propriété de vivacité bornée qui énonce un délai maximal avant que la « situation souhaitée » ne finisse par avoir lieu.
- **Les propriétés d'équité.** Ce type de propriété se rapproche beaucoup du type énoncé pour les propriétés de vivacité. Nous énonçons, dans le cas présent, que sous certaines conditions quelque chose aura lieu (ou pas) une infinité de fois. Le terme « équité » signifie une répartition équitable.

Nous pouvons signaler une propriété particulière, **l'absence de blocage**, qui énonce que le système ne se trouve jamais dans une situation où il est impossible de progresser. En logique temporelle, l'absence de blocage s'écrit $AG\ EX\ true$, ce qui signifie « quel que soit l'état atteint, il existera un état successeur immédiat ».

3.1.4 Présentation de l'outil de *model-checking* UppAal

Les algorithmes de model-checking sont implémentés dans des logiciels. Une importante activité de recherche concerne ces outils et vise à augmenter leur efficacité et à repousser leurs limites. Il existe plusieurs *model-checkers* temporisés : UppAal, Kronos [BDM+98], HyTech [HHWT95] par exemple. L'outil HYTECH (pour HYbrid TECHnology Tool) a été développé à l'université de Berkeley par Tom Henzinger, Pei-Hsin Ho et Howard Wong-Toi. Cet outil vise à vérifier des réseaux d'automates hybrides très généraux. Kronos, quant à lui, vérifie des automates à l'aide de propriétés exprimées à l'aide de *TCTL*.

L'outil UppAal se démarque des deux autres outils par son interface graphique très conviviale. Son module de simulation est très performant et permet, lors de la phase de modélisation, de faire des tests du modèle pour détecter d'éventuelles erreurs dans la modélisation. UppAal est un outil de *model checking* pour les systèmes temps réels. Il est développé conjointement par les universités d'Uppsala et d'Aalborg et c'est en 1995 que la première version fut disponible [LPY97]. Depuis, l'outil est en perpétuelle évolution grâce à un développement constant comme

le montrent de nombreux travaux [ABB+01, BBD+02, DBLY02, DBLY03] et les nombreuses études de cas qui ont été réalisées, comme [HSLL97, LP97, LPY98, BGK+96]. Tout cela atteste de la maturité et de la qualité de cet outil.

3.1.4.1 On modélise ...

Dans UppAal, un système est décrit par un réseau d'automates temporisés (*TA*) (Définition 3.3) étendus à l'aide de variables entières, de types de données structurés et de la synchronisation de canaux.

Lors de la phase de modélisation, la définition du système se décompose en trois parties :

– *La déclaration de variables globales* : Il est possible de définir des variables globales et des fonctions qui sont accessibles et partagées par tous les *TA*. UppAal accepte la définition de tableaux de variables, de canaux ou d'horloges ainsi que la définition de nouveaux types. La figure 3.3 présente un exemple d'une déclaration de variables globales d'un système UppAal avec la définition d'une horloge h, d'un nouveau type `id_Q`, tel que si i est une variable de type `id_Q` alors $1 \leq i \leq 3$, et d'un tableau de 3 canaux de synchronisation nommés `x[1]`, `x[2]` et `x[3]`.

```
1  clock h; // définition d'une horloge
2  typedef int[1,3] id_Q; // définition d'un nouveau type
3  chan x[id_Q]; // définition d'un tableau de canaux
```

FIGURE 3.3 – Déclaration de variables globales dans UppAal.

– *La définition de modèles* : Dans un système UppAal, les *TA* sont définis comme des instances de modèle. Dans la définition de ces modèles de *TA*, il est possible d'ajouter des déclarations de variables locales ou des paramètres. Ces paramètres sont alors considérés comme des variables locales aux instances de *TA* et sont initialisés lors de la déclaration du système. La figure 3.4 présente la définition de deux modèles de *TA* ayant chacun un paramètre.

```
1  process P(const int v) {...}
2
3  process Q(id_Q a) {...}
```

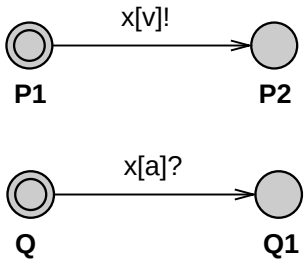


FIGURE 3.4 – Définition des deux modèles d'automate temporisés UppAal.

– *La déclaration du système* : La déclaration du système consiste à la définition d'un ou plusieurs processus concurrents, chacun modélisé par un *TA*. Un processus est une instance d'un

modèle de *TA*. L'instanciation d'un modèle d'automate demande de lier une variable à chaque paramètre défini par le modèle. Si le paramètre n'est pas défini par un type borné alors le paramètre doit être lié explicitement. Dans le cas contraire, où le paramètre est d'un type borné, il est possible de laisser UppAal lier automatiquement une instance du modèle avec chaque valeur du type défini comme paramètre. La figure 3.5 présente les cas de figure avec la déclaration d'un système de quatre processus :

– Un processus défini par le modèle `P` et modélisé par le *TA* nommé `P(2)`
– Trois processus définis par le modèle `Q` et modélisé par les *TA* nommés `Q(1)`, `Q(2)`, `Q(3)`.

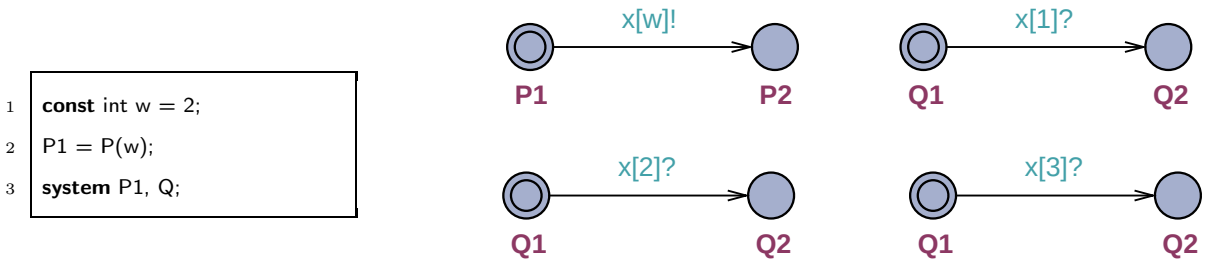


Figure 3.5 – Déclaration d'un système UppAal de quatre automates temporisés.

Parmi les différents éléments qui composent un système défini dans UppAal, nous listons ceux que nous utilisons dans la suite de ce chapitre :

– *les actions de synchronisation non-déterministes.* Il est possible de définir des actions de synchronisation non-déterministes via la définition d'un tableau de canaux associé à la déclaration d'une plage d'identifiants propre à une transition. Dans ce cas, une action de synchronisation se situe dans une plage d'actions liées à la plage d'identifiants définie. Par exemple, dans la figure 3.6 la transition entre les états `P1` et `P2` définit un identifiant `a` de type entier avec $1 \leq a \leq 2$. Les actions de synchronisation possibles sont alors `x[1]?` et `x[2]?`.

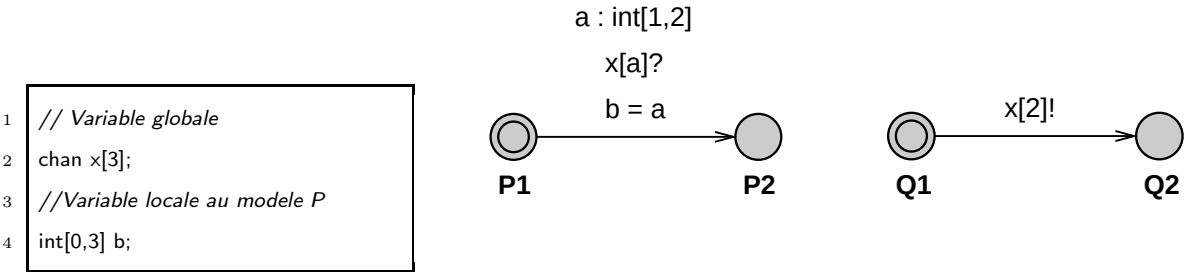


Figure 3.6 – Exemple d'utilisation d'une action de synchronisation non-déterministe.

– *les canaux de diffusion.* Si un canal, noté `c` (cf. Figure 3.7), est déclaré `broadcast`, alors une action d'émission sur ce canal, notée `c!`, déclenche toutes les actions de réception sur ce canal, notées `c?`. Une action d'émission sur un canal de diffusion n'est pas bloquante : une émission sur le canal `c` n'impose pas qu'il y ait une action de réception sur un canal défini par un autre *TA*.

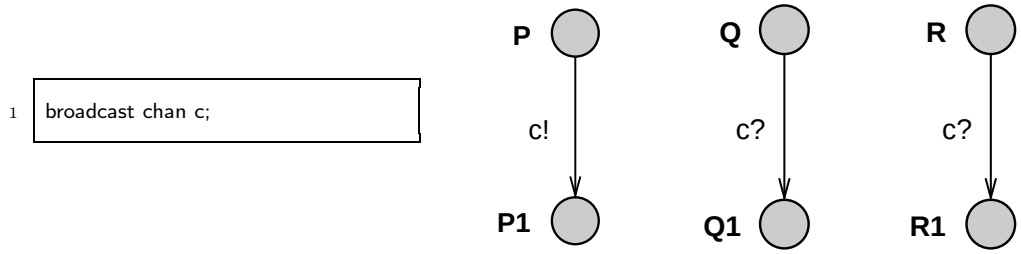
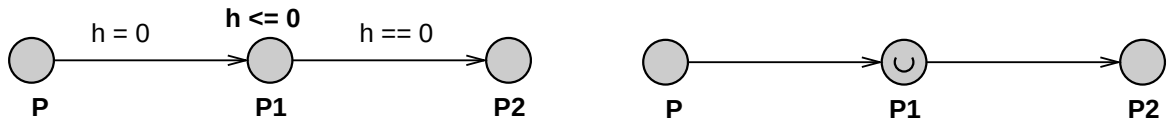


FIGURE 3.7 – Exemple d'un canal de diffusion. Le canal `c` est défini comme un canal de diffusion.

– *les localités urgentes.* Si une localité est déclarée *urgente* alors le temps ne peut pas s'écouler dans cette localité. Définir une localité l urgente est équivalent à définir une horloge h qui est remise à zéro par toutes les transitions arrivant dans la localité l et définir un invariant $h \leq 0$ pour la localité l. Par la suite, nous modélisons une localité *urgente* par un cercle avec la lettre majuscule U (U comme $Urgent$). Par exemple, dans la figure 3.8b, la localité `P1` est définie comme une localité urgente et le comportement de l'automate est équivalent à celui de la figure 3.8a.



(a) Sans localité urgente.

(b) Avec localité urgente (`P1`).

FIGURE 3.8 – Exemple d'un automate avec une localité urgente.

– *les localités atomiques.* Une localité dite *committed* est une localité *urgente* qui impose que la prochaine transition du système soit une transition sortante de cette localité (ou d'une autre localité *committed*). Ce type de localité permet de modéliser une suite de transitions atomiques [7]. Par la suite, nous modélisons une localité *committed* par un cercle avec la lettre majuscule C (C comme $Committed$). Par exemple, dans la figure 3.9, la localité `P1` est définie comme une localité *committed*

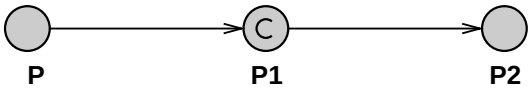


FIGURE 3.9 – Exemple d'un automate avec une localité *committed*.

Formellement, un automate temporisé est défini comme suit :

Définition 3.8 (Automate Temporisé UPPAAL**)** *Un automate temporisé $\mathcal{A}$ est un 8-uplet $\langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$ où :*
– L est un ensemble fini de localités ;

7. Une suite atomique de transitions garantit que toutes les transitions sont effectuées sans être interrompues, ou qu'aucune transition n'est effectuée.

- $l_0 \in L$ *est la localité initiale ;*
- V *est un ensemble de variables,*
- C *est un ensemble d'horloges* ($C \cap V = \emptyset$)*,*
- $E \subseteq L \times \mathcal{G}(C,V) \times Sync \times Act \times L$ *est un ensemble fini de transitions où* $\mathcal{G}(C,V)$ *est un ensemble de contraintes autorisées dans les gardes,* $Sync$ *est un ensemble d'actions (de synchronisation ou interne),* Act *est un ensemble d'actions d'affection et de réinitialisation d'horloges.*
- $Assign \subseteq Act$ *est un ensemble d'affectations qui affectent des valeurs initiales à des variables,*
- $L \rightarrow Inv(C,V)$ *associe un invariant à chaque localité ;*
- $K_L : L \rightarrow \{o,u,c\}$ *affecte un type (ordinary, urgent, committed) à chaque localité.*

Le tuple $\langle l_1, Select, Guards, action, Assign, l_2 \rangle$ définit une transition entre les localités l_1 et l_2 où $Select$ est un ensemble de variables locales à la transition, $Guards$ est un ensemble de gardes, $action$ est une action de synchronisation et $Assign$ est un ensemble d'affectations de variables ou d'appels de fonctions. Par exemple, les deux transitions modélisées par la figure 3.6 sont $\langle P1, \{a\}, \emptyset, x[a]?, \{b := a\}, P2 \rangle$ et $\langle Q1, \emptyset, \emptyset, x[2]!, \emptyset, Q2 \rangle$. A partir de la définition 3.8, il est possible de définir un réseau d'automates temporisés UppAal :

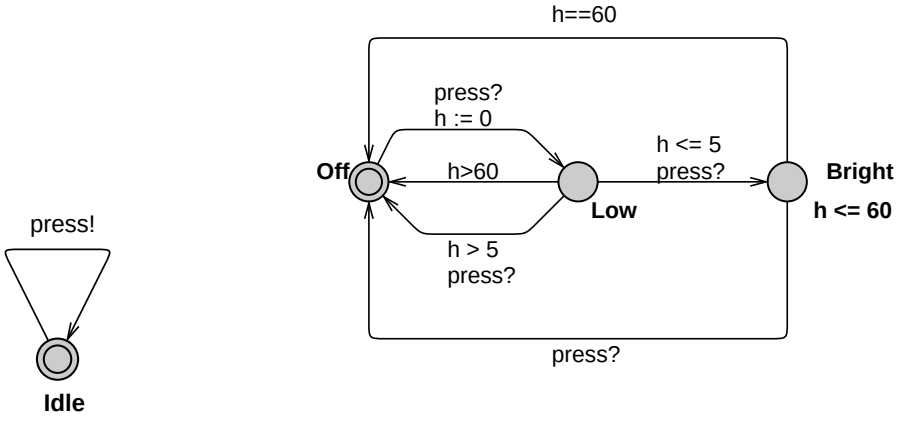
Définition 3.9 (Réseau d'Automates Temporisés UppAal) *Un réseau d'automates temporisés* UppAal *est un 7-uplet* $\langle \vec{A}, \vec{l_0}, V_g, C_g, Ch, K_{Ch}, Assign_g \rangle$ *où :*
- $\vec{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ *est un vecteur de n automates temporisés* $\mathcal{A}_i = \langle L_i, l_i^0, E_i, V_i, C_i, Assign_i, Inv_i, K_i^L \rangle$
- $\vec{l_0} = (l_1^0, \ldots, l_n^0)$ *est le vecteur des localités initiales,*
- V_g *est un ensemble de variables globales partagées par tous les automates temporisés* $\mathcal{A}_i$*,*
- C_g *est un ensemble d'horloges globales partagées par tous les automates temporisés* $\mathcal{A}_i$ ($C_g \cap V_g = \emptyset$)*,*
- Ch *est un ensemble de canaux utilisées pas les automates temporisés* $\mathcal{A}_i$ *pour communiquer entre eux* ($C_g \cap Ch = \emptyset$ *et* $V_g \cap Ch = \emptyset$)*,*
- $K_{Ch} : Ch \rightarrow \{o,u\}$ *affecte un type (ordinary ou urgent) à chaque canal.*
- $Affect_g$ *est un ensemble d'affectations qui affectent des valeurs initiales à des variables globales.*

Nous rappelons que tous les automates temporisés dans UppAal sont des instances de modèle. La définition 3.10 ajoute la notion de paramètre à la définition 3.8 afin de pouvoir déclarer des modèles d'automate.

Définition 3.10 (Modèle d'Automates Temporisés UppAal) *Un modèle d'automate temporisé* $\mathcal{M}$ *est noté* $\mathcal{M}(x_1, \ldots, x_n) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$ *où :*
- $\langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$ *est un un 8-uplet identique à celui donné par la définition 3.8 ;*
- $x_1, \ldots, x_n$ *est la liste des paramêtres du modèle. Ces paramêtres peuvent être des variables, des horloges ou des canaux ;*

Une instance de modèle est notée $\mathcal{A} = \mathcal{M}(y_1, \ldots, y_n)$ où chaque y_i remplace le x_i correspondant défini pour le modèle $\mathcal{M}$.

La figure 3.10 présente la représentation graphique de deux modèles d'automate temporisé à l'aide de l'outil UppAal. Le modèle *Switch* pour l'automate de la figure 3.1a et le modèle *Lamp* pour l'automate de la figure 3.2.



(a) *Switch* (b) *Lamp*

FIGURE 3.10 – Modélisation de deux modèles d'automate temporisé représentant un interrupteur (cf. Figure 3.1a) et une lampe (cf. Figure 3.2) à l'aide de l'outil UppAal.

3.1.4.2 ... puis on vérifie

UppAal permet de vérifier des propriétés d'atteignabilité (cf. Section 3.1.3.2) sur les réseaux d'automates temporisés :

- $A.e$ exprime que l'automate A est dans l'état e
- $v \sim n$ (v étant une variable) ou $h \sim n$ (h étant une horloge) avec $n \in \mathbb{N}$ et $\sim$ parmi les symboles de comparaison $=$, $<$, $>$, $\leq$ et $\geq$.

Le langage de requêtes utilisé pour spécifier les propriétés à vérifier est un sous-ensemble de TCTL (cf. section 3.1.3.2). En considérant que ϕ et ψ sont deux propositions atomiques, nous listons les différentes formes que peut prendre une propriété qui sera vérifiée par UppAal :

- $A[]\ \varphi$ « pour tous les chemins et pour tous les états, φ est valide »
- $E<>\ \varphi$ « il existe un chemin où éventuellement, φ est valide »
- $A<>\ \varphi$ « pour tous les chemins, éventuellement φ est valide »
- $E[]\ \varphi$ « il existe un chemin ou pour tous les états, φ est valide »
- $\psi -->\ \varphi$ « ψ mène toujours à φ »

Par exemple, nous pouvons tester un système comprenant deux automates temporisés issus des deux modèles proposés par la figure 3.10 à l'aide des propriétés suivantes :

- « `A[] Switch.Idle` » : L'automate *Switch* se trouve toujours dans l'état *Idle*

– « `E<> Lamp.Low and h > 60` » : Il est possible que l'automate *Lamp* se trouve dans l'état *Low* alors que l'horloge h est supérieure à 60.
– « `A[] Lamp.Bright imply not h>60` » : Le fait que l'automate *Lamp* soit dans l'état *Bright* implique que l'horloge h ne peut pas être supérieure à 60.

Notons, pour finir, la possibilité de vérifier qu'il n'existe pas d'état bloquant dans le système avec la propriété « `A[] not deadlock` ».

3.2 Transformation d'un système π-calcul vers un réseau d'automates

Dans cette section, nous définissons la transformation d'un système défini dans un algèbre de processus (en l'occurrence en π-calcul polyadique d'ordre supérieur présenté et utilisé dans le chapitre 2) en un système d'automates. Nous ajoutons ensuite la notion de temps à l'aide des automates temporisés afin de rendre ce système d'automates tolérant aux pannes inhérentes d'un système distribué.

Malgré sa simplicité syntaxique, le π-calcul est doté d'une grande richesse sémantique. La modélisation dans cet algèbre de processus conduit à la définition de systèmes infinis. Nous pouvons distinguer deux types de systèmes infinis [PP06] :

– **Les systèmes à nombre d'états infinis.** Les opérations de préfixe, de choix et de récursivité fournies par le π-calcul sont équivalentes au formalisme des automates finis. Si la récursion seule ne conduit pas à des systèmes infinis, l'ajout de l'opérateur parallèle modifie cette propriété. En effet cette situation permet la création dynamique de processus qui, par conséquence, aboutit à des systèmes infinis.
– **Les systèmes paramétrés.** Lors de la définition d'un système, son environnement n'est pas connu. Il doit par conséquent être un système ouvert. Le fait de paramétrer un système conduit à la possibilité d'avoir un nombre d'environnement potentiellement infini.

Ces deux types de systèmes offerts par le π-calcul font apparaitre les premières limites d'une transformation d'une spécification π-calcul vers un système d'automates. La définition d'un agent π-calcul définissant une récursivité associée à du parallélisme ne peut être transformée en un automate. De plus, l'environnement lors de l'invocation d'un agent doit être connu car la sémantique des automates ne permet pas la création d'automates paramétrables [8]. Par conséquent, la définition d'un automate ne peut se faire sans prendre en compte le réseau dans lequel il se trouve.

Nos travaux décrits dans cette section sont proches des travaux réalisés par [FGMM03]. Les auteurs y proposent la transformation d'agents π-calcul en automates. Cette transformation est possible en utilisant des « *HD-Automata* » dans lesquels les états et les étiquettes sont enrichis d'une liste de noms. Ce type de transformation permet de vérifier un système défini en

8. En tout cas dans le sens défini pour les systèmes paramétrés en π-calcul. La suite de cette section propose néanmoins de paramétrer les automates, résultats de la transformation, par un identifiant.

π-calcul en utilisant l'environnement *HAL* (pour *HD Automata Laboratory*) qui offre un module de vérification par model-checking de propriétés exprimées à l'aide d'une logique temporelle.

Nous reprenons cette notion de transformation d'un système défini en π-calcul vers un système d'automates en vue de vérifier des propriétés exprimées à l'aide d'une logique temporelle avec, dans notre cas, celle définie par l'outil UppAal. Un premier travail dans ce sens a été réalisé dans notre équipe par A. Barbu et F. Mourlin [BM02], ce que nous continuons par la suite avec la définition de règles de transformation.

3.2.1 Définitions des différents opérateurs de transformation

Pour présenter les différentes opérateurs et règles de transformation que nous définissons dans les sections 3.2.1 et 3.2.2, nous prenons l'exemple très simple d'une spécification π-calcul composée de la définition de deux agents, A et B, et d'un terme principal $System$ (eq. 3.1).

$$\begin{array}{l} System \stackrel{def}{=} (\nu\, y)\, \Big(A(\, y\,) \mid B(\, y\,) \Big) \\ \text{avec} \left\{ \begin{array}{lcl} A(\, x\,) & \stackrel{def}{=} & (\nu\, u)\, \Big(\overline{x}\langle\, u, v\,\rangle\, .\, \Big(u(\, b\,)\, .\, 0 \Big) + \Big(v(\, c\,)\, .\, A(\, c\,) \Big) \Big) \\ B(\, z\,) & \stackrel{def}{=} & z(\, w, v\,)\, .\, \Big(\Big((\nu\, a)\, \overline{w}\langle\, a\,\rangle\, .\, 0 \Big) + \Big((\nu\, d)\, \overline{v}\langle\, d\,\rangle\, .\, B(\, d\,) \Big) \Big) \end{array} \right. \end{array} \tag{3.1}$$

Le système modélisé par cette spécification met en parallèle deux agents, un agent défini par A et un agent défini par B, les deux instances étant paramétrées par le nom y. En suivant la sémantique opérationnelle du π-calcul présentée dans la section 2.1.1.3, nous pouvons écrire qu'un agent défini par $A(x)$ suit la réduction suivante :

$$A \xrightarrow{\overline{x}\langle u,v\rangle} A' \xrightarrow{v(c)} A(\, c\,) \;\text{ avec } A' \stackrel{def}{=} \Big(u(\, b\,)\, .\, 0 \Big) + \Big(v(\, c\,)\, .\, A(\, c\,) \Big)$$

et un agent défini par $B(z)$ suit la réduction duale suivante :

$$B \xrightarrow{z(w,v)} B' \xrightarrow{\overline{v}\langle d\rangle} B(\, d\,) \;\text{ avec } B' \stackrel{def}{=} \Big((\nu\, a)\, \overline{w}\langle\, a\,\rangle\, .\, 0 \Big) + \Big((\nu\, d)\, \overline{v}\langle\, d\,\rangle\, .\, B(\, d\,) \Big)$$

Il est alors facile d'imaginer une transformation « brute » de ces deux définitions d'agent en deux automates. La figure 3.11 propose le résultat que pourrait donner une telle transformation : chaque définition d'agent π-calcul est transformée en un automate où les réductions successives sont modélisées par des localités et les préfixations des processus sont modélisées par des transitions.

L'utilisation du résultat de cette transformation nous est alors difficile car l'outil UppAal, présenté en 3.1.4, n'est pas capable d'exploiter ces deux automates en vue de vérifier des propriétés exprimées à l'aide d'une logique temporelle.

Notre travail consiste donc à définir la transformation d'une spécification π-calcul vers un réseau

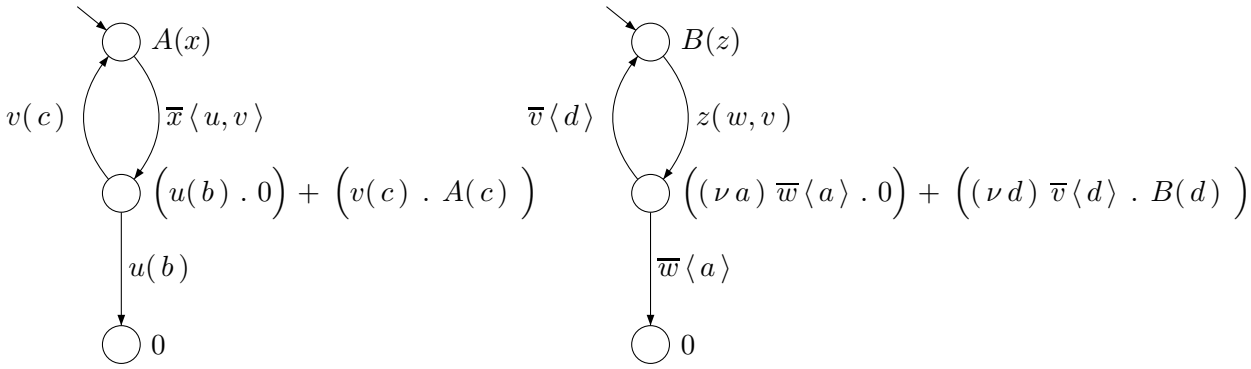


FIGURE 3.11 – Résultat de la transformation « brute » des définitions d'agent π-calcul (eq. 3.1) en automates.

d'automates temporisés. Celui-ci doit être exploitable par l'outil UPPAAL. Cette transformation, d'une spécification π-calcul vers un système UPPAAL, est possible grâce à la définition des trois opérateurs de transformation suivants :

– l'opérateur $\mathcal{T}_\alpha$ (déf. 3.11) qui transforme un agent π-calcul en un modèle d'automate ;
– l'opérateur $\mathcal{T}_\pi$ (déf. 3.13) qui transforme la définition d'un terme π-calcul en un modèle d'automate ;
– l'opérateur $\mathcal{T}_S$ (déf. 3.14) qui transforme une spécification π-calcul en un système UPPAAL.

Comme toute définition d'un système UPPAAL (cf. Section 3.1.4), le résultat de la transformation d'une spécification π-calcul par l'opérateur $\mathcal{T}_S$ définit un ensemble de processus, chacun modélisé par un automate, mis en parallèle. Ces processus sont des instances de modèles d'automate. Chaque modèle d'automate définit alors un paramètre formel de type entier permettant d'associer à chaque processus un identifiant. Par exemple, la notation $\mathcal{A} = \mathcal{M}(id_A)$ définit un *TA* comme instance du modèle d'automate $\mathcal{M}$, ayant un identifiant nommé id_A. Cet identifiant est alors utilisé par la règle pour définir la transition $start[id]?$.

Pour la définition de nos opérateurs de transformation (et des règles de transformations associées), nous adoptons la convention d'écriture suivante :
– les modèles d'automates sont notés $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \ldots$
– les *TA*, qui représentent les processus du système, sont notés $\mathcal{A}, \mathcal{B}, \ldots$ avec, par exemple, $\mathcal{A} = \mathcal{M}_1(id_A), \mathcal{B} = \mathcal{M}_3(id_B)$

3.2.1.1 Transformation d'un agent π-calcul

Le premier opérateur de transformation que nous présentons est celui qui réalise la transformation d'un agent π-calcul en un modèle d'automate (déf. 3.11). On parle alors de l'α-transformation (se lit « alpha-transformation ») de l'agent P vers le modèle d'automate $\mathcal{M}$. Le résultat de cette

transformation, noté $\mathcal{M}'$, est un modèle d'automate temporisé possédant le même paramètre formel *id* et la même localité initiale que $\mathcal{M}$.

Définition 3.11 (α-transformation) *Une α-transformation associe un triplet composé d'un agent π-calcul, d'un modèle d'automate* UppAal *source et d'un état appartenant à ce modèle d'automate, à un couple composé d'un modèle d'automate* UppAal *cible et d'un état appartenant à ce modèle d'automate.*
Le domaine de définition D d'une α-transformation est défini par

$$\mathcal{T}_\alpha : D \subset \mathbb{P} \times \mathbb{A} \times \mathbb{Q} \rightarrow \mathbb{A} \times \mathbb{Q}$$

où :
– $\mathbb{P}$ *est l'ensemble des agents π-calcul ;*
– $\mathbb{A}$ *est l'ensemble des modèles d'automate ;*
– $\mathbb{Q}$ *est l'ensemble des localités définies par les modèles d'automate ;*

Remarque 3.1 (Syntaxe d'une α-transformation) *Une α-transformation est notée*

$$\mathcal{T}_\alpha(P,\, \mathcal{M}(id),\, l_f) = (\mathcal{M}'(id), l'_f)$$

où :
– P *est un agent π-calcul ;*
– $\mathcal{M}(id)$ *est un modèle d'automate* UppAal *tel que* $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$ *;*
– $l_f \in L$ *modélise l'état du système avant l'évaluation de l'agent P ;*
– $\mathcal{M}'(id)$ *est un modèle d'automate* UppAal *tel que* $\mathcal{M}'(id) = \langle L', l_0, E', V', C', Assign', Inv', K'_L \rangle$.
– $l'_f \in L'$ *modélise l'état du système après l'évaluation de l'agent P ;*

Pour chaque comportement d'agent défini par la syntaxe du π-calcul (cf. Tableau 2.1), nous définissons une règle de transformation pour l'opérateur $\mathcal{T}_\alpha$. La section 3.2.2 présente toutes les règles de transformation correspondant aux différents comportements que peut prendre un agent π-calcul.

La définition 3.12 détermine la composition de deux α-transformations. Ce type de composition n'est possible que si les deux α-transformations sont appliquées sur le même modèle d'automate, c'est à dire que le deuxième argument des deux opérateurs $\mathcal{T}_\alpha$ est identique.

Définition 3.12 (Composition d'α-transformations) *Soient deux α-transformations définies pour les agents P_1 et P_2 sur le modèle d'automate $\mathcal{M}(id)$ alors la composée de ces deux α-transformations est définie par*

$$\Big(\mathcal{T}_\alpha(P_2) \circ \mathcal{T}_\alpha(P_1)\Big)\Big(\mathcal{M}(id), l_f\Big) = \mathcal{T}_\alpha(P_2, \mathcal{M}'(id), l'_f)$$

avec $\mathcal{T}_\alpha(P_1, \mathcal{M}(id), l_f) = (\mathcal{M}'(id), l'_f)$

3.2.1.2 Transformation d'une définition d'agent π-calcul

La définition 3.11 fixe à trois le nombre de paramètres pour l'opérateur de transformation $\mathcal{T}_\alpha$: l'agent π-calcul à transformer, un modèle d'automate et une localité appartenant à ce modèle. Il est donc nécessaire de définir un opérateur de transformation qui soit capable de créer un modèle d'automate. C'est le rôle de l'opérateur de transformation $\mathcal{T}_\pi$. Celui-ci transforme la définition d'un agent π-calcul en un modèle d'automate (déf. 3.13). On parle alors de la π-transformation (se lit « pi-transformation ») d'une définition d'un agent P en un modèle d'automate $\mathcal{M}$. Lors de la transformation d'une spécification π-calcul chaque définition d'agent est transformée par l'opérateur $\mathcal{T}_\pi$ en un modèle d'automate.

Définition 3.13 (π-transformation) *Une π-transformation, notée $\mathcal{T}_\pi$, associe un agent π-calcul à un modèle d'automate* UppAal.
Le domaine de définition D d'une π-transformation est défini par

$$\mathcal{T}_\alpha : D \subset \mathbb{P} \rightarrow \mathbb{A}$$

où :
- $\mathbb{P}$ *est l'ensemble des agents π-calcul ;*
- $\mathbb{A}$ *est l'ensemble des modèle d'automate ;*

Remarque 3.2 (Syntaxe d'une π-transformation) *Une π-transformation est notée*

$$\mathcal{T}_\pi(A) = \mathcal{M}(id)$$

où :
- *A est la définition d'un agent π-calcul tel que* $A \stackrel{def}{=} P$ *avec P représentant un processus ;*
- $\mathcal{M}(id)$ *est un modèle d'automate* UppAal *possédant un paramètre formel, nommé id, de type entier.*

Dans la section 3.2.2, nous définissons deux règles de transformation pour l'opérateur $\mathcal{T}_\pi$: la règle 3.2 pour la transformation d'une définition d'agent de type $A(x_1, \ldots, x_n) \stackrel{def}{=} P$ et la règle 3.1 pour la transformation d'un terme principal *System*.

3.2.1.3 Transformation d'une spécification π-calcul

L'opérateur de transformation d'une spécification π-calcul en un réseau d'automates temporisés UppAal (déf. 3.9) est noté $\mathcal{T}_S$ (déf. 3.14). L'application de cet opérateur a pour but de définir :
- *Un contexte de transformation* : celui-ci est utilisé lors de l'application des opérateurs $\mathcal{T}_\pi$ et $\mathcal{T}_\alpha$. Il se traduit par la déclaration :

  - d'une variable, nommée *name_counter*, et d'une fonction *succ* qui incrémente la valeur de la variable donnée en paramètre et retourne sa nouvelle valeur. La fonction *succ* et la variable *name_counter* sont utilisées conjointement lors de l'α-transformation d'une création de noms (cf. Règle 3.7) ;
  - d'une constante, nommée *NULL*, utilisée par les modèles de *TA* lors de la transformation d'une émission ou d'une réception (cf. Section 3.2.2.5) ;
  - de tableaux de canaux *com*, *comHO*, *start*, *finish*, *param*. Ces derniers sont utilisés lors de la définition des modèles de *TA* ;
- *Un système* UppAal : la définition d'un système UppAal consiste à déclarer une liste de m processus concurrents, chaque processus étant modélisé par un *TA* noté $\mathcal{A}_j$ tel que $\mathcal{A}_j = \mathcal{M}_i(id_j)$ avec $1 \leq j \leq m$ et $\mathcal{M}_i(id)$ un modèle d'automate résultat de la π-transformation d'une définition d'agent de la spécification π-calcul avec $1 \leq i \leq n$ et n le nombre de définitions d'agent dans la spécification π-calcul ($m \leq n$).

Définition 3.14 *[Opérateur de transformation $\mathcal{T}_S$] L'opérateur de transformation $\mathcal{T}_S$ transforme une spécification π-calcul vers un réseau d'automates temporisés* UppAal. *L'opérateur $\mathcal{T}_S$ est défini par*

$$\mathcal{T}_S(\pi Spec) = \langle \vec{\mathcal{A}}, I\vec{dle}, V_g, \emptyset, Ch, K_{Ch}, Assign_g \rangle$$

où :

- *$\pi Spec$ est une spécification π-calcul composée de la définition d'un terme principal, noté $System$, et de n définitions d'agent identifiées par la notation $A_1(\vec{x_1}), A_2(\vec{x_2}), \ldots, A_n(\vec{x_n})$ où les vecteurs $\vec{x_1}, \vec{x_2}, \ldots, \vec{x_n}$ définissent respectivement les paramètres formels de chaque définition d'agent.*
- *m le nombre d'invocations d'agent, notés $A_i(\vec{y_i})$ où les vecteurs $\vec{y_i}$ définissent les paramètres effectifs de chaque invocation avec $1 \leq i \leq n$;*
- *$\vec{\mathcal{A}} = (\mathcal{S}, \mathcal{A}_1, \ldots, \mathcal{A}_m)$ un vecteur d'automates temporisés où $\mathcal{S} = \mathcal{T}_\pi(System)$ pour le terme principal et $\mathcal{A}_j = \mathcal{M}_i(id_j)$ pour chaque invocation avec $1 \leq i \leq n$ et $1 \leq j \leq m$ où pour chaque définition d'un agent $A_i(\vec{x_i})$ il existe un modèle d'automate $\mathcal{M}_i(id)$ tel que $\mathcal{M}_i(id) = \mathcal{T}_\pi(A_i)$ et $i \leq j$;*
- *$I\vec{dle} = (Idle_S, Idle_1, \ldots, Idle_m)$ est le vecteur des localités initiales de chaque TA du système ;*
- *$V_g = \{name\_counter, NULL\}$ est l'ensemble des variables et constantes globales du système ;*
- *$Ch = \{param[m][p], com[p][p], comHO[p][m], start[m], finish[m]\}$, avec p le nombre total de noms, est l'ensemble de canaux utilisés par les automates $\mathcal{A}_j$;*
- *$Assign_g = \{name\_counter := 0, NULL := -1\}$ est l'ensemble des affectations ;*
- *$\forall\, c \in Ch$, $K_{Ch}(c) = o$*

La figure 3.28 présente les déclarations globales (cf. Figure 3.12a) et la définition (cf. Figure 3.12c) d'un système UppAal obtenues par l'application de l'opérateur $\mathcal{T}_S$ sur la spécification définie par l'équation 3.1. Notons, dans la figure 3.12a, la définition de trois constantes (l. 1 à 3). Les

constantes `NB_AGENTS` et `NB_NAMES` sont utilisées pour définir les types `agent` (l. 5) et `name` (l. 6). Ces derniers permettent de borner les tableaux de canaux `start`, `finish`, `com`, `param`, `comHO` (l. 8 à 12) et de déclarer la variable `name_counter` (l. 13). La valeur de cette variable est incrémentée par l'appel de la fonction `succ` (l. 15) (cf. Règle 3.7). Enfin, les déclarations globales contiennent la définition de deux nouveaux types, `id_P` et `id_Q` (l. 17 et 18), qui sont utilisés pour définir respectivement le paramètre du modèle de *TA* nommé `P` et le paramètre du modèle de *TA* nommé `Q` (cf. Figure 3.12b). Le système défini dans la figure 3.12c contient donc trois processus : une instance du modèle `System`, une instance du modèle `P` avec l'identifiant 1 et une instance du modèle `Q` avec l'identifiant 2.

```
1  const int NB_AGENTS = 2;
2  const int NB_NAMES = 100;
3  const int NULL = −1;
4
5  typedef int[1, NB_AGENTS] agent;
6  typedef int[0, NB_NAMES] name;
7
8  chan start[agent];
9  chan finish[agent];
10 chan com[name][name];
11 chan param[agent][name];
12 chan comHO[name][agent];
13 name name_counter = 0;
14
15 name succ(name &x){ return ++x; }
16
17 typedef int[1,1] id_P ;
18 typedef int[2,2] id_Q ;
```

(a) Déclarations globales

```
1  process System() {...}
2
3  process P(const id_P id) {...}
4
5  process Q(const id_Q id) {...}
```

(b) Définition des modèles de *TA*

```
1  system System, P, Q;
```

(c) Définition du système

FIGURE 3.12 – Résultat dans UPPAAL de l'application de l'opérateur $\mathcal{T}_S$ sur la spécification π-calcul définie par l'équation 3.1.

3.2.2 Définitions des règles de transformation

Après avoir défini les trois opérateurs de transformation permettant la transformation d'une spécification π-calcul en un système UPPAAL (cf. Section 3.2.1), nous présentons dans cette section les règles de transformation qui s'appliquent pour deux de ces opérateurs :

– les règles qui s'appliquent à l'opérateur $\mathcal{T}_\pi$ concernant la définition d'agent π-calcul (Section 3.2.2.1) ;

– les règles qui s'appliquent à l'opérateur $\mathcal{T}_\alpha$ en fonction du comportement d'un agent : la restriction de noms (Section 3.2.2.4), l'invocation d'agent (Section 3.2.2.2), la composition parallèle (Section 3.2.2.3), la préfixation (Section 3.2.2.5), les communications d'ordre supérieur (Section 3.2.2.6), la récursion (Section 3.2.2.7) ou la somme (Section 3.2.2.8).

3.2.2.1 Définition d'un agent

A partir de la définition 3.13 de l'opérateur $\mathcal{T}_\pi$, nous spécifions deux règles de transformation : la règle 3.1 pour la transformation du terme principal *System* et la règle 3.2 pour la définition d'un agent notée $A(x_1, \ldots, x_n)$.

L'application de l'opérateur de transformation $\mathcal{T}_\pi$ sur un terme principal, noté $System \stackrel{def}{=} P$, d'une spécification π-calcul respecte la règle 3.1. Le résultat de cette π-transformation est le résultat de l'α-transformation de l'agent P appliquée sur un modèle d'automate possédant une seule localité, notée $Idle$.

Règle 3.1 *[Transformation du terme principal] Soit* $System \stackrel{def}{=} P$ *la définition du terme principal d'une spécification* π*-calcul alors* $\mathcal{T}_\pi(System) = \mathcal{T}_\alpha(P, \mathcal{M}(id), Idle)$ *avec* $\mathcal{M}(id)$ *un modèle d'automate tel que*

$$\mathcal{M}(id) = \langle \{Idle\}, Idle, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, K_L \rangle \text{ où } K_L(Idle) = o.$$

Le terme principal d'une spécification π-calcul, noté *System*, permet de déterminer l'invocation des différents agents qui sont exécutés en parallèle. Nous pouvons observer ceci dans l'équation 3.1. Le système UppAal, résultat de la transformation d'une spécification par l'opérateur $\mathcal{T}_S$, définit une unique instance du modèle d'automate $\mathcal{M}(id)$ tel que $\mathcal{T}_\pi(System) = \mathcal{M}(id)$. La localité $Idle$ de ce modèle d'automate est la localité initiale du réseau d'automates temporisés défini par le système UppAal.

De son coté, la règle 3.2 définit une transformation lorsque l'opérateur de transformation $\mathcal{T}_\pi$ est appliqué sur la définition d'un agent notée $A(x_1, \ldots, x_n) = P$ avec n le nombre de paramètres formels. Le résultat de cette π-transformation est le résultat de l'α-transformation de l'agent P vers un modèle d'automate possédant au moins deux localités : une localité initiale, notée $Idle$, et une localité, notée *Started*. S'ajoutent à ces deux localités, n localités $Init_i (i = 1, \ldots, n)$ en fonction du nombre de paramètres formels de la définition de l'agent correspondant.

Règle 3.2 *[Transformation d'une définition d'un agent] Soit* $A(x_1, \ldots, x_n) \stackrel{def}{=} P$ *la définition d'un agent* π*-calcul avec* $x_1, \ldots, x_n$ *en tant que paramètres formels alors*

$$\mathcal{T}_\pi(A(x_1, \ldots, x_n)) = \mathcal{T}_\alpha(P, \mathcal{M}(id), Started)$$

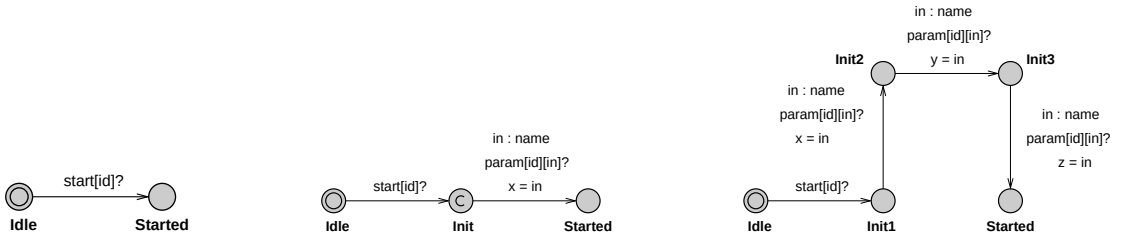
avec $\mathcal{M}(id)$ un modèle d'automate définissant un paramètre formel, nommé id, de type entier tel que

$$\mathcal{M}(id) = \langle \{Idle, Started\} \cup L, Idle, E, V, \emptyset, \emptyset, \emptyset, K_L \rangle$$

où $K_L(Idle) = K_L(Started) = o$. Si n est l'arité de A alors :
- *Si $\boxed{n = 0}$ alors $L = \emptyset$, $V = \emptyset$ et $E = \{\langle Idle, \emptyset, \emptyset, start[id]?, \emptyset, Started\rangle\}$*
- *Si $\boxed{n = 1}$ alors $L = \{Init\}$, $V = \{x\}$ et $K_L(Init) = c$*
  et $E = \{\langle Idle, \emptyset, \emptyset, start[id]?, \emptyset, Init\rangle, \langle Init, \{in\}, \emptyset, param[id][in]?, \{x := in\}, Started\rangle\}$
- *Si $\boxed{n > 1}$ alors $L = \{Init_1, \ldots, Init_n\}$, $V = \{x_1, \ldots, x_n\}$, $K_L(Init_i) = o$, $i = 1, \ldots, n$*
  et $E = \{\langle Idle, \emptyset, \emptyset, start[id]?, \emptyset, Init_1\rangle,$
  $\langle Init_n, \{in\}, \emptyset, param[id][in]?, \{x_n := in\}, Started\rangle\} \cup E'$
  où $E' = \{e_1, \ldots, e_{n-1}\}$ et $e_i = \langle Init_i, \{in\}, \emptyset, param[id][in]?, \{x_i := in\}, Init_{i+1}\rangle$

L'application de la règle 3.2 fait intervenir deux tableaux de canaux déclarés dans la définition 3.14 : les tableaux *start* et *param*. Chaque modèle de *TA* issu de cette transformation possède une unique transition sortante de sa localité *Idle*. Cette transition est étiquetée par l'action de réception $start[id]?$ avec *id* défini comme paramètre formel pour le modèle de *TA*. Le tableau *param* est lui utilisé pour initialiser les paramètres formels définis pour l'agent *A*. Chaque paramètre formel x_i est initialisé par une transition étiquetée par l'action de réception $param[id][x_i]?$. Ces transitions modélisent le début de l'évaluation d'un agent et à un autre automate de pouvoir démarrer cet automate. Cette technique se rapproche du concept de « *Callable Timed Automata* » présenté dans [BVBF13]. Les transitions étiquetées par les actions d'émission duales sont créées lors de l'application de la règle de transformation définie pour l'invocation d'un agent (cf. Règle 3.4). La figure 3.13 présente le résultat de trois π-transformations en fonction du nombre de paramètres formels défini pour le terme π-calcul.

Une autre démarche serait de définir la valeur des différents paramètres à l'aide de variables globales au système UppAal. Notre choix s'est porté sur la définition de variables locales pour respecter la portée des variable comme le permet le pi-calcul. De plus, l'utilisation de variables globales entrainerait une perte de lisibilité du système UppAal résultat de la transformation.



(a) $\mathcal{T}_\pi(A)$ (b) $\mathcal{T}_\pi(A(x))$ (c) $\mathcal{T}_\pi(A(x, y, z))$

FIGURE 3.13 – Exemples de π-transformations en fonction du nombre de paramètres formels.

La transition étiquetée $start[id]?$ modélise le déclenchement de l'évaluation de l'agent π-calcul représenté par le modèle de *TA* contenant cette transition. La fin de l'évaluation d'un agent π-

calcul est représentée par l'action nulle, notée 0. La règle 3.3 définit l'α-transformation de cette action nulle. Cette règle utilise le tableau de canaux $finish$ déclaré dans la définition 3.14 et ajoute une transition sortante à la localité représentant l'état de l'agent π-calcul avant la fin de son évaluation. Cette transition est étiquetée par l'action d'émission $finish[id]!$ avec id défini comme paramètre formel pour le modèle de *TA*. La transition étiquetée par l'action de réception duale est créée lors de l'application de la règle de transformation lors de l'exécution d'un agent (cf. Règle 3.5).

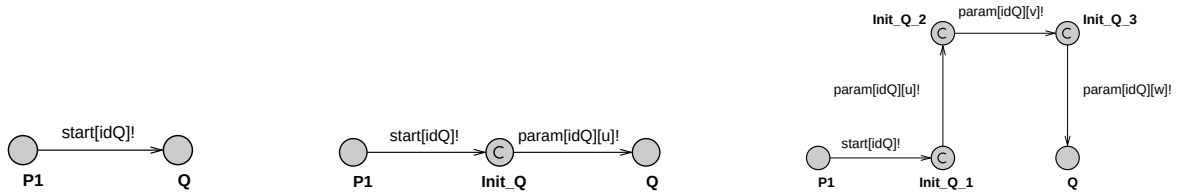
Règle 3.3 *[Transformation d'une action nulle] Soit $A(x_1, \ldots, x_n)$ la définition d'un agent π-calcul et $\mathcal{M}$ un modèle d'automate tel que $\mathcal{T}_\pi(A(x_1, \ldots, x_n)) = \mathcal{M}(id)$*
avec $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$. Si l_f représente l'état d'un agent A avant une action nulle, notée 0*, alors*

$$\mathcal{T}_\alpha(0,\, \mathcal{M}(id),\, l_f) = \Big(\langle L \cup \{Finished\}, l_0, E \cup E_\alpha, V, C, Assign, Inv, K_{L_\alpha} \circ K_L \rangle, Finished\Big)$$

avec $E_\alpha = \{\langle l_f, \emptyset, \emptyset, finish[id]!, \emptyset, Finished \rangle\}$, $K_{L_\alpha}(l_f) = c$ et $K_{L_\alpha}(Finished) = o$

3.2.2.2 Exécution d'un agent

En π-calcul, l'exécution d'un agent consiste à démarrer son évaluation. Celle-ci est réalisée tant qu'une action nulle n'est pas évaluée. La section 3.2.2.1 fixe que chaque modèle de *TA* résultant de l'application de la règle 3.2 possède une unique transition sortante de sa localité *Idle* (cf. Figure 3.13). Cette transition est étiquetée par l'action de réception $start[id]?$ et modélise le début de l'évaluation de l'agent correspondant. Nous avons aussi défini que l'α-transformation d'une action nulle ajoute une transition étiquetée par l'action d'émission $finish[id]!$ au modèle de *TA* (cf. Règle 3.3). L'α-transformation de l'exécution d'un agent doit donc définir au moins deux transitions : une transition étiquetée par l'action d'émission $start[id_A]!$ et une transition étiquetée par l'action de réception $finish[id_A]?$ avec id_A la valeur de l'identifiant du *TA* modélisant l'agent à exécuter.



(a) $\mathcal{T}^s_\alpha(A, \mathcal{M}_1(id), P1)$ (b) $\mathcal{T}^s_\alpha(A(u), \mathcal{M}_1(id), P1)$ (c) $\mathcal{T}^s_\alpha(A(u, v, w), \mathcal{M}_1(id), P1)$

FIGURE 3.14 – Modélisations d'α-transformations d'une invocation d'agent A vers un modèle d'automate $\mathcal{M}_1$ en fonction de son nombre de paramètres formels.

La règle 3.5 définit l'α-transformation de l'exécution d'un agent, notée $A(y_1, \ldots, y_n)$, comme la composition de deux α-transformations (déf. 3.12). Par exemple, pour transformer l'exécution

d'un agent $A(y_1, \ldots, y_n)$ modélisé par le *TA* nommé $\mathcal{A}$ tel que $\mathcal{A} = \mathcal{M}(id_A)$ et $\mathcal{M}$ le modèle de *TA* résultat de la π-transformation de $A(x_1, \ldots, x_n)$ (cf. Règle 3.2), la composition d'α-transformations se structure de la façon suivante :

- d'une α-transformation qui suit la règle 3.4, notée $\mathcal{T}_\alpha^s$, et permet de modéliser l'invocation de l'agent π-calcul. Cela consiste à définir une transition étiquetée par l'action d'émission $start[id_A]!$ pour modéliser le déclenchement de l'évaluation de A et n transitions étiquetées par les actions d'émission $param[id_A][x_i]!$ pour modéliser l'initialisation des n paramètres formels de A.
- et d'une α-transformation qui suit la règle 3.5, notée $\mathcal{T}_\alpha^f$, et permet de modéliser la fin de l'évaluation de l'agent π-calcul. Cela consiste à ajouter une transition étiquetée par l'action de réception $finish[id_A]?$.

La figure 3.14 présente le résultat de l'application de la règle 3.4 en fonction du nombre de paramètres effectifs à fournir lors de l'invocation de l'agent A.

Règle 3.4 *[Transformation de l'invocation d'un agent] Soient deux π-transformations telles que $\mathcal{T}_\pi(A_1) = \mathcal{M}_1(id)$ avec $\mathcal{M}_1(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$ et $\mathcal{T}_\pi(A_2(x_1, \ldots, x_n)) = \mathcal{M}_2(id)$. Si l_f représente l'état d'un agent A_1 avant l'instanciation de l'agent $A_2(y_1, \ldots, y_n)$, modélisé par l'automate $\mathcal{A}_2$ tel que $\mathcal{A}_2 = \mathcal{M}_2(id_{A_2})$, alors*

$$\mathcal{T}_\alpha^s(A_2(y_1, \ldots, y_n),\, \mathcal{M}_1(id),\, l_f) = \Big(\mathcal{M}_1'(id), l_f'\Big)$$

avec

$$\mathcal{M}_1'(id) = \langle L \cup \{l_f'\} \cup L_\alpha, l_0, E \cup E_\alpha, V \cup \{id_{A_2}\} \cup V_\alpha, C, Assign, Inv, K_{L_\alpha} \circ K_L \rangle$$

où $K_{L_\alpha}(l_f) = K_{L_\alpha}(l_f') = o$ et n l'arité du terme A_2 alors

- *Si* $\boxed{n = 0}$ *alors $L_\alpha = \emptyset$, $V_\alpha = \emptyset$ et $E_\alpha = \{\langle l_f, \emptyset, \emptyset, start[id_{A_2}]!, \emptyset, l_f' \rangle\}$*
- *Si* $\boxed{n = 1}$ *alors $L_\alpha = \{Init\}$, $V_\alpha = \{y\}$ et $K_{L_\alpha}(Init) = c$*
  et $E_\alpha = \{\langle l_f, \emptyset, \emptyset, start[id_{A_2}]!, \emptyset, Init \rangle, \langle Init, \emptyset, \emptyset, param[id_{A_2}][y]!, \emptyset, l_f' \rangle\}$
- *Si* $\boxed{n > 1}$ *alors $L_\alpha = \{Init_1, \ldots, Init_n\}$, $V = \{y_1, \ldots, y_n\}$ et $K_{L_\alpha}(Init_i) = c$, $i = 1, \ldots, n$*
  et $E = \{\langle l_f, \emptyset, \emptyset, start[id_{A_2}]!, \emptyset, Init_1 \rangle, \langle Init_n, \emptyset, \emptyset, param[id_{A_2}][y_n]!, \emptyset, l_f' \rangle\} \cup E_\alpha'$
  où $E_\alpha' = \{e_1, \ldots, e_{n-1}\}$ et $e_i = \langle Init_i, \emptyset, \emptyset, param[id_{A_2}][y_{i+1}]!, \emptyset, Init_{i+1} \rangle$

Règle 3.5 *[Transformation d'une exécution d'un agent] Soient deux π-transformations telles que $\mathcal{T}_\pi(A_1) = \mathcal{M}_1(id)$, $\mathcal{T}_\pi(A_2(x_1, \ldots, x_n)) = \mathcal{M}_2(id)$ et une α-transformation $\mathcal{T}_\alpha^s(A_2(y_1, \ldots, y_n),\, \mathcal{M}_1(id),\, l_f) = (\mathcal{M}_1'(id), l_f^s)$ avec $\mathcal{M}_1'(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$. Si l_f représente l'état d'un agent A_1 avant l'exécution de l'agent $A_2(y_1, \ldots, y_n)$, modélisé par l'automate $\mathcal{A}_2$ tel que $\mathcal{A}_2 = \mathcal{M}_2(id_{A_2})$, alors*

$$\begin{aligned} \mathcal{T}_\alpha(A_2(y_1, \ldots, y_n),\, \mathcal{M}_1(id),\, l_f) &= \Big(\mathcal{T}_\alpha^f\big(A_2(y_1, \ldots, y_n)\big) \circ \mathcal{T}_\alpha^s\big(A_2(y_1, \ldots, y_n)\big)\Big)\Big(\mathcal{M}_1(id),\, l_f\Big) \\ &= (\mathcal{M}_1''(id), l_f'') \end{aligned}$$

avec

$\mathcal{M}_1''(id) = \langle L \cup \{l_f''\}, l_0, E \cup \{\langle l_f^s, \emptyset, \emptyset, finish[id_{A_2}]?, \emptyset, l_f''\rangle\}, V \cup \{id_{A_2}\}, C, Assign, Inv, K_{L_\alpha} \circ K_L\rangle$

où $K_{L_\alpha}(l_f^s) = K_{L_\alpha}(l_f'') = o$

3.2.2.3 Composition parallèle

En π-calcul, la notation $P \mid Q$ met en parallèle les agents P et Q. Il est alors possible d'écrire la mise en parallèle de n agents par la notation $A_1 \mid \cdots \mid A_n$. La règle 3.6 définit la transformation de ce type de notation. Lors d'une telle transformation, nous pouvons distinguer deux étapes :

- L'invocation des n d'agents A_i avec $i = 1, \ldots, n$. Il s'agit de la composition des n α-transformations qui suivent la règle 3.4. L'ensemble de ces invocations doit être réalisé de manière atomique, c'est pourquoi les localités communes représentant la fin de l'invocation d'un agent A_i et le début de l'invocation d'un agent A_{i+1} sont des localités atomiques : $K_L(l_{f_i}) = c$ pour $i = 1, \ldots, n-1$.
- La fin d'évaluation des n agents invoqués précédemment. Celle-ci est réalisée par l'α-transformation notée $\mathcal{T}_\alpha^f(A_1 \mid \cdots \mid A_n,\ \mathcal{M}, l_f)$. Cette α-transformation ajoute au modèle d'automate deux transitions : la transition $\langle l_f, \{in\_id\}, \emptyset, finish[in\_id]?, \{finished := finished+1\}, l_{f_n}\rangle$ permettant de compter le nombre d'agents arrêtés (à l'aide de la variable *finished*) et la transition $\langle l_f, \emptyset, \{finished == n\}, \emptyset, \{finished := 0\}, l_f'\rangle$ modélisant la fin de l'exécution de tous les agents

Un exemple de résultat de ce type d'α-transformation est donnée dans la figure 3.15 lors de la π-transformation du terme *System* (eq. 3.1).

Règle 3.6 *[La composition parallèle] Soit $A(x_1, \ldots, x_n)$ la définition d'un terme π-calcul et $\mathcal{M}$ un modèle d'automate tel que $\mathcal{T}_\pi(A) = \mathcal{M}(id)$ avec $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L\rangle$. Si l_f modélise l'état de A avant l'exécution parallèle de n processus, notés $A_1, \ldots, A_n$, alors*

$$\mathcal{T}_\alpha(A_1 \mid \cdots \mid A_n,\ \mathcal{M}(id),\ l_f) = \Big(\mathcal{T}_\alpha^f(A_1 \mid \cdots \mid A_n) \circ \mathcal{T}_\alpha^s(A_n) \circ \cdots \circ \mathcal{T}_\alpha^s(A_1)\Big)\Big(\mathcal{M}(id),\ l_f\Big)$$

avec

- $\mathcal{T}_\alpha^s(A_i,\ \mathcal{M}(id),\ l_f) = \Big(\langle L_i, l_0, E_i, V_i, C_i, Assign_i, Inv_i, K_{L_i}\rangle, l_{f_i}\Big)$ *pour* $i = 1, \ldots, n$;
- $l_{f_i} \in L_i$ *représentant l'état de A après l'invocation de* A_i*. Cet état est obtenu par l'*α*-transformation de* A_i *suivant la règle 3.4 (cet état est alors noté* l_f'*)* ;
- $K_L(l_{f_i}) = c$ *pour* $i = 1, \ldots, n-1$;
- $\mathcal{T}_\alpha^f(A_1 \mid \cdots \mid A_n,\ \mathcal{M}(id),\ l_f) = (\mathcal{M}'(id), l_f')$ *avec* $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L\rangle$ *et* $\mathcal{M}'(id) = \langle L \cup \{l_f'\}, l_0, E \cup E_\alpha, V \cup \{finished\}, C, Assign, Inv, K_{L_\alpha} \circ K_L\rangle$ *où* $E_\alpha = \{\langle l_f, \{in\_id\}, \emptyset, finish[in\_id]?, \{finished := finished+1\}, l_{f_n}\rangle,$ $\langle l_f, \emptyset, \{finished == n\}, \emptyset, \{finished := 0\}, l_f'\rangle\}$ *et* $K_{L_\alpha}(l_f) = K_{L_\alpha}(l_f') = o$.

3.2.2.4 Restriction de nom

La notation $(\nu\, x_1, \ldots, x_n)\, P$ offerte par le π-calcul permet de restreindre n noms, notés $x_1, \ldots, x_n$, à un agent P. L'α-transformation de cette notation suit la règle 3.7. Celle-ci définit la composition de deux α-transformations : l'α-transformation de l'agent P précédée de l'α-transformation de la notation $(\nu\, x_1, \ldots, x_n)$. Cette dernière se traduit par la déclaration d'une variable locale au modèle de *TA* pour chaque nouveau nom. Il est important que chaque variable locale à un *TA* représentant un nom π-calcul ait une valeur unique dans le système UPPAAL auquel appartient le TA. Cette unicité est obtenue à l'aide de la variable globale *name_counter* résultant de l'application l'opérateur $\mathcal{T}_S$ (déf. 3.14). La valeur affectée à chaque variable locale représentant un nom π-calcul est le résultat de l'appel de la fonction *succ* sur la variable globale *name_counter*. La fonction *succ* incrémente la valeur de la variable passée en paramètre et retourne sa nouvelle valeur. L'appel *succ*(*name_counter*) pour chaque nom permet alors de compter le nombre de noms créés et de récupérer un identifiant unique pour chacun. L'affectation de chacune des variables locales par un identifiant unique est réalisée lors d'une transition. L'α-transformation de la notation $(\nu\, x_1, \ldots, x_n)$ définit alors une transition correspondant aux n affectations de variable, chacune réalisant un appel de *succ*(*name_counter*).

Règle 3.7 *[Transformation d'une création de nom] Soit $x_1, \ldots, x_n$ un ensemble de n noms restreints à un processus P, noté $Q \stackrel{def}{=} (\nu\, x_1, \ldots, x_n)\, P$, alors la transformation de cette restriction vers un modèle d'automate $\mathcal{M}(id)$, avec $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$, est une composition de deux α-transformations telles que*

$$\mathcal{T}_\alpha\Big(Q,\, \mathcal{M}(id),\, l_f\Big) = \mathcal{T}_\alpha\Big(P,\, \mathcal{M}'(id),\, l'_f\Big)$$

avec

$$\mathcal{M}'(id) = \langle L \cup \{l'_f\}, l_0, E \cup \{e\}, V \cup V_\alpha, C, Assign, Inv, K_{L_\alpha} \circ K_L \rangle$$

où $K_{L_\alpha}(l_f) = K_{L_\alpha}(l'_f) = o$ et :
- *Si* $\boxed{n = 1}$ *alors $V_\alpha = \{x\}$ et $e = \langle l_f, \emptyset, \emptyset, \emptyset, \{x := succ(name\_counter)\}, l'_f \rangle$*
- *Si* $\boxed{n > 1}$ *alors $V_\alpha = \{x_1, \ldots, x_n\}$*
  et $e = \langle l_f, \emptyset, \emptyset, \emptyset, \{x_1 := succ(name\_counter), \ldots, x_n := succ(name\_counter)\}, l'_f \rangle$

A ce stade, nous avons défini toutes les règles de transformation permettant de réaliser la π-transformation du terme *System* (eq. 3.1). En effet, l'application des règles 3.1, 3.7 et 3.6 donne la π-transformation du terme principal (cf. Règle 3.1) :

$$\mathcal{T}_\pi(System) = \mathcal{T}_\alpha((\nu\, x)\ (P(x) \mid Q(x)), \mathcal{M}(id), Idle\Big)$$

avec $\mathcal{M}(id) = \langle \{Idle\}, Idle, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, K_L \rangle$ et *Init* une localité appartenant au modèle de *TA* résultat de l'α-transformation $\mathcal{T}_\alpha((\nu\, x)\ , \mathcal{M}, Idle)$. La figure 3.15 présente le modèle de *TA* obtenu par cette π-transformation.

La section suivante présente les règles de transformation appliquées à la préfixation d'agent. La définition de ces règles permet de réaliser les π-transformations des définitions des agents $A(y)$ et $B(y)$ de l'équation 3.1 utilisé par le terme principal *System*.

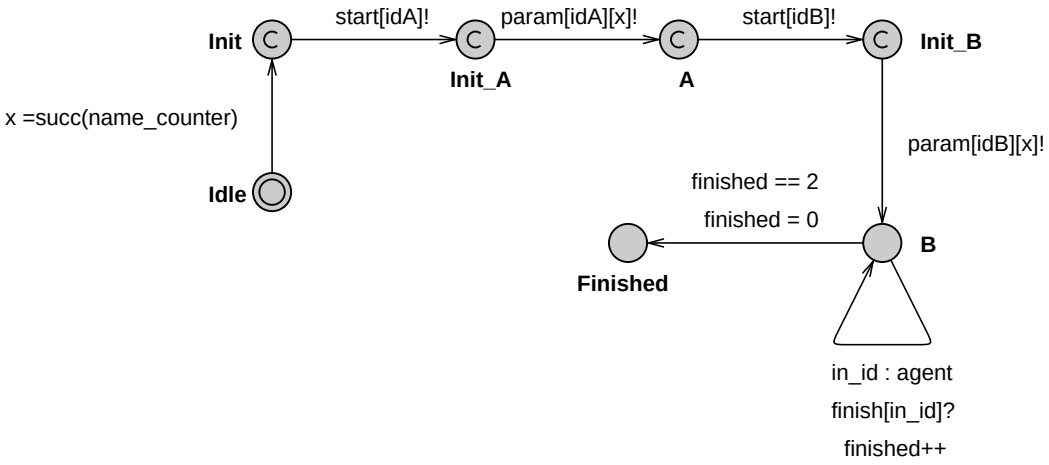


FIGURE 3.15 – Modélisation de la π-transformation du terme *System* (eq. 3.1).

3.2.2.5 Préfixation d'agents

L'α-transformation d'une préfixation d'agent, notée $\alpha.P$, suit la règle 3.8. Cette règle est définie par la composition de deux α-transformations : l'α-transformation de l'agent P précédée de l'α-transformation de l'action α. Pour chaque forme que peut prendre l'action α, nous définissons une règle de transformation : la règle 3.9 pour l'action d'émission π-calcul, la règle 3.10 pour l'action de réception π-calcul et la règle 3.11 pour l'action interne.

Règle 3.8 *[Transformation d'une préfixation d'un processus] Soit $A(x_1, \ldots, x_n)$ la définition d'un agent π-calcul et $\mathcal{M}$ un modèle d'automate tel que $\mathcal{T}_\pi(A(x_1, \ldots, x_n)) = \mathcal{M}(id)$*
avec $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$. Si l_f modélise l'état d'un agent A avant la préfixation d'un processus P par une action α, notée $\alpha \,.\, P$ alors

$$\mathcal{T}_\alpha(\, \alpha \,.\, P,\, \mathcal{M}(id),\, l_f) = \Big(\mathcal{T}_\alpha(P) \,\circ\, \mathcal{T}_\alpha(\alpha)\Big)\Big(\mathcal{M}(id),\, l_f\Big) =$$

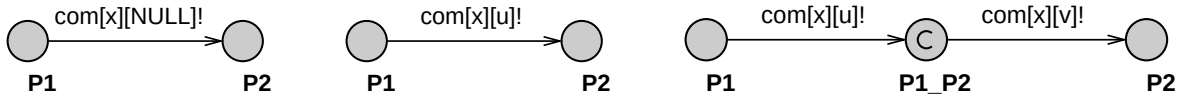
L'action d'émission

L'α-transformation d'une communication entre deux agents π-calcul utilise le tableau de canaux *com* déclaré dans la définition 3.14. Les deux dimensions de ce tableau sont bornées par le type *name* (cf. Section 3.2.1.3). Par exemple, l'émission à travers un canal x d'un nom u, notée $\overline{x}\,\langle\, u \,\rangle$ en π-calcul, est alors modélisée par une transition étiquetée par l'action d'émission $com[x][u]!$ avec x et u deux variables de type *name* locales au modèle de *TA* représentant l'agent qui émet sur le canal x. La règle 3.9 qui définit l'α-transformation d'une émission utilise ce type

de canaux. L'émission $\overline{x}\langle u \rangle$, que nous prenons comme exemple, est celle qui est définie par le π-calcul monadique (cf. Section 2.1.1.1).

La règle 3.9 définit aussi l'α-transformation d'une émission polyadique (cf. Section 2.1.1.5). Dans ce cas, pour une émission notée $\overline{x}\langle u, v \rangle$, il y a deux transitions étiquetées par une action d'émission : $com[x][u]!$ et $com[x][v]!$ avec x, u et v des variables locales au modèle de *TA* représentant l'agent qui émet sur le canal x. Ces deux transitions doivent être passées de manière atomique, la localité commune entre ces deux transitions est définie comme *committed*.

La figure 3.16 présente les résultats de trois α-transformations suivant la règle 3.9 en fonction du nombre de noms envoyés par un canal x. Notons la présence d'un troisième cas d'émission, avec la notation $\overline{x}$, et l'utilisation de la constante globale `NULL` déclarée dans la définition 3.14 car le tableau de canaux `com` a deux dimensions.



(a) $\mathcal{T}_\alpha(\overline{x}\,, \mathcal{M}(id), P1)$ (b) $\mathcal{T}_\alpha(\overline{x}\langle u \rangle\,, \mathcal{M}(id), P1)$ (c) $\mathcal{T}_\alpha(\overline{x}\langle u, v \rangle\,, \mathcal{M}(id), P1)$

FIGURE 3.16 – Modélisations d' α-transformations d'une action d'émission définie par le règle 3.9.

Règle 3.9 *[Transformation d'une émission] Soit $A(x_1, \ldots, x_n)$ la définition d'un agent π-calcul et $\mathcal{M}$ un modèle d'automate tel que $\mathcal{T}_\pi(A(x_1, \ldots, x_n)) = \mathcal{M}(id)$*
avec $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$. Si l_f modélise l'état d'un agent A avant une action d'émission à travers le canal a de n noms, notés $x_1 \ldots x_n$, alors

$$\mathcal{T}_\alpha(\overline{a}\langle x_1, \ldots, x_n \rangle\,, \mathcal{M}(id), l_f) = \left(\mathcal{M}'(id), l'_f\right)$$

avec

$$\mathcal{M}'(id) = \langle L \cup \{l'_f\} \cup L_\alpha, l_0, E \cup E_\alpha, V \cup \{a\} \cup V_\alpha, C, Assign, Inv, K_{L_\alpha} \circ K_L \rangle$$

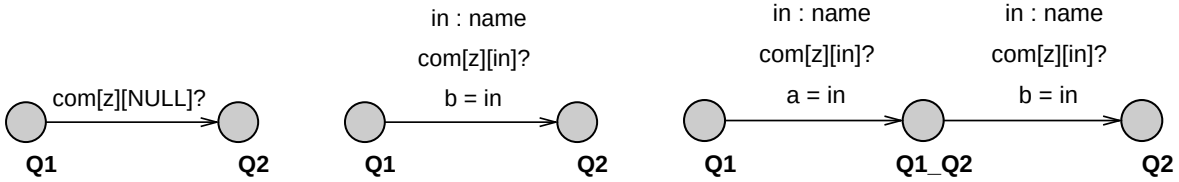
où $K_{L_\alpha}(l_f) = K_{L_\alpha}(l'_f) = o$ et :
- *Si $\boxed{n = 0}$ alors $L_\alpha = \emptyset$, $V_\alpha = \emptyset$ et $E_\alpha = \{\langle l_f, \emptyset, \emptyset, com[a][NULL]!, \emptyset, l'_f \rangle\}$*
- *Si $\boxed{n = 1}$ alors $L_\alpha = \emptyset$, $V_\alpha = \{x\}$ et $E = \{\langle l_f, \emptyset, \emptyset, com[a][x]!, \emptyset, l'_f \rangle\}$*
- *Si $\boxed{n > 1}$ alors $L_\alpha = \{l_{f_1}, \ldots, l_{f_{n-1}}\}$, $V_\alpha = \{x_1, \ldots, x_n\}$ et $K_{L_\alpha}(l_{f_i}) = c,\ i = 1, \ldots, n-1$ et $E_\alpha = \{\langle l_f, \emptyset, \emptyset, com[a][x_1]!, \emptyset, l_{f_1} \rangle, \langle l_{f_{n-1}}, \emptyset, \emptyset, com[a][x_n]!, \emptyset, l'_f \rangle\} \cup E'_\alpha$ où $E'_\alpha = \{e_1, \ldots, e_{n-2}\}$ et $e_i = \langle l_{f_i}, \emptyset, \emptyset, com[a][x_i]!, \emptyset, l_{f_{i+1}} \rangle$*

L'action de réception

La règle de transformation duale de la règle 3.9 est la règle de transformation 3.10. Celle-ci définit l'α-transformation d'une réception π-calcul. Ainsi, comme pour l'émission π-calcul, la

règle 3.10 liste trois cas en fonction du nombre de noms reçus par le canal de réception. L'α-transformation d'une réception utilise donc le tableau de canaux *com*, le même que celui utilisé par l'α-transformation d'une émission. Dans le cas d'une réception, nous utilisons la possibilité de pouvoir déclarer une variable locale à une transition (cf. Section 3.1.4.1). Par exemple, la réception à travers un canal z d'un nom b, notée $z(b)$ en π-calcul, est modélisée par une transition définie par le tuple $\langle l_1, \{in\}, \emptyset, com[z][in]?, \{b := in\}, l_2\rangle$ avec in une variable de type *name* locale à cette transition, z et b deux variables de type *name* locales au modèle de *TA* représentant l'agent qui reçoit sur le canal z. Cette transition affecte enfin la valeur de in à la variable b.

La figure 3.17 présente les résultats de trois α-transformations suivant la règle 3.10 en fonction du nombre de noms reçus par un canal x.



(a) $\mathcal{T}_\alpha(\, z,\, \mathcal{M}(id),\, Q1)$ (b) $\mathcal{T}_\alpha(\, z(b),\, \mathcal{M}(id),\, Q1)$ (c) $\mathcal{T}_\alpha(\, z(a,b),\, \mathcal{M}(id),\, Q1)$

FIGURE 3.17 – Modélisations d' α-transformations d'une action de réception définie par le règle 3.10.

Règle 3.10 *[Transformation d'une réception] Soit $A(x_1, \ldots, x_n)$ la définition d'un agent π-calcul et $\mathcal{M}$ un modèle d'automate tel que $\mathcal{T}_\pi(A(x_1, \ldots, x_n)) = \mathcal{M}(id)$*
avec $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L\rangle$. Si l_f modélise l'état d'un agent A avant une action de réception à travers le canal a de n noms, notés $x_1 \ldots x_n$, alors

$$\mathcal{T}_\alpha(\, a(\, x_1, \ldots, x_n\,)\ ,\ \mathcal{M}(id),\, l_f) = \Big(\mathcal{M}'(id), l'_f\Big)$$

avec

$$\mathcal{M}' = \langle L \cup \{l'_f\} \cup L_\alpha, l_0, E \cup E_\alpha, V \cup \{a\} \cup V_\alpha, C, Init, Inv, K_{L_\alpha} \circ K_L\rangle$$

où $K_{L_\alpha}(l_f) = K_{L_\alpha}(l'_f) = o$ et :
- *Si $\boxed{n = 0}$ alors $L_\alpha = \emptyset$, $V_\alpha = \emptyset$ et $E_\alpha = \{\langle l_f, \emptyset, \emptyset, com[a][NULL]?, \emptyset, l'_f\rangle\}$*
- *Si $\boxed{n = 1}$ alors $L_\alpha = \emptyset$, $V_\alpha = \{a\}$ et $E_\alpha = \{\langle l_f, \{in\}, \emptyset, com[a][in]?, \{x := in\}, l'_f\rangle\}$*
- *Si $\boxed{n > 1}$ alors $L_\alpha = \{l_{f_1}, \ldots, l_{f_{n-1}}\}$, $V_\alpha = \{x_1, \ldots, x_n\}$ et $K_{L_\alpha}(l_{f_i}) = o,\ i = 1, \ldots, n-1$ et $E_\alpha = \{\langle l_f, \{in\}, \emptyset, com[a][in]?, \{x_1 := in\}, l_{f_1}\rangle, \langle l_{f_{n-1}}, \{in\}, \emptyset, com[a][in]?, \{x_n := in\}, l_f\rangle\} \cup E'_\alpha$ où $E'_\alpha = \{e_1, \ldots, e_{n-2}\}$ et $e_i = \langle l_{f_i}, \{in\}, \emptyset, com[a][in]?, \{x_i := in\}, l_{f_{i+1}}\rangle$*

Nous présentons les α-transformations des communications d'ordre supérieur dans la section 3.2.2.6.

L'action interne

L'action π-calcul notée τ est une action interne au processus dans lequel elle est déclarée. L'α-transformation de cette notation suit la règle 3.11. Celle-ci transforme une action interne π-calcul en une transition non étiquetée entre une localité représentant l'état du processus avant l'action interne et une localité représentant l'état du processus après cette action interne.

Règle 3.11 *[Transformation d'une action interne] Soit $A(x_1, \ldots, x_n)$ la définition d'un agent π-calcul et $\mathcal{M}$ un modèle d'automate tel que $\mathcal{T}_\pi(A(x_1, \ldots, x_n)) = \mathcal{M}(id)$*
avec $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$. Si l_f modélise l'état d'un agent A avant une action interne, notée τ, alors

$$\mathcal{T}_\alpha(\,\tau,\, \mathcal{M}(id),\, l_f) = \Big(\langle L \cup \{l'_f\}, l_0, E \cup \{\langle l_f, \emptyset, \emptyset, \emptyset, \emptyset, l'_f \rangle\}, C, Assign, Inv, K_{L_\alpha} \circ K_L \rangle, l'_f\Big)$$

où $K_{L_\alpha}(l_f) = K_{L_\alpha}(l'_f) = o$

3.2.2.6 Communications d'ordre supérieur

Nous avons présenté, dans la section 2.1.2, le π-calcul d'ordre supérieur qui permet de faire transiter des processus par des canaux de communication. Par exemple, dans le système défini par le terme *System* de l'équation 3.2, l'agent A envoie à travers le canal z le processus C à l'agent B.

$$System \stackrel{def}{=} (\,\nu\, x\,)\,\Big(A(\,x\,) \mid B(\,x\,)\Big) \text{ avec } \begin{cases} C(\,w\,) \stackrel{def}{=} \tau\,.\,C'\,.\,0 \\ A(\,z\,) \stackrel{def}{=} (\,\nu\, y\,)\,\overline{z}\,\langle\, C(y)\,\rangle\,.\,A' \\ B(\,x\,) \stackrel{def}{=} x(\,Q(\,u\,)\,)\,.\,Q(\,u\,)\,.\,B' \end{cases} \tag{3.2}$$

La transformation de ce type de communication demande de définir deux nouvelles règles d'α-transformation : la règle 3.12 pour l'émission de termes d'ordre supérieur et la règle 3.13 pour la réception de termes d'ordre supérieur. Ces deux règles utilisent deux tableaux de canaux déclarés dans la définition 3.14 : le tableau *comHO* pour communiquer l'identifiant du TA modélisant le terme d'ordre supérieur à transiter et le tableau *param* pour communiquer la valeur de ses paramètres effectifs. La règle 3.12 ajoute au modèle de TA auquel elle s'applique au moins une transition étiquetée par une action d'émission appartenant au tableau *comHO* pour envoyer l'identifiant du TA modélisant le terme d'ordre supérieur. S'ajoutent à cette transition, n transitions (et n localités) étiquetées par une action d'émission utilisant le tableau *param* correspondant aux n paramètres formels définis pour le terme d'ordre supérieur.

Règle 3.12 *[Transformation d'une émission d'ordre supérieur] Soient deux π-transformations telles que $\mathcal{T}_\pi(A_1) = \mathcal{M}_1(id)$ avec $\mathcal{M}_1(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$*

et $\mathcal{T}_\pi(A_2(x_1,\ldots,x_n)) = \mathcal{M}_2(id)$.
Si l_f *représente l'état d'un agent* A_1 *avant une action d'émission à travers le canal* a *de l'agent* $A_2(y_1,\ldots,y_n)$*, modélisé par l'automate* $\mathcal{A}_2$ *tel que* $\mathcal{A}_2 = \mathcal{M}_2(id_{A_2})$*, alors*

$$\mathcal{T}_\alpha(\,\overline{a}\,\langle\, A_2(y_1,\ldots,y_n)\,\rangle\ ,\ \mathcal{M}_1(id),\, l_f) = \Big(\mathcal{M}'_1(id), l'_f\Big)$$

$$\mathcal{M}'_1(id) = \langle L \cup \{l'_f\} \cup L_\alpha, l_0, E \cup E_\alpha, V \cup \{id_{A_2}, a\} \cup V_\alpha, C, Assign, Inv, K_{L_\alpha} \circ K_L\rangle$$

où $K_{L_\alpha}(l_f) = K_{L_\alpha}(l'_f) = o$ *et* n *l'arité d'un agent* A_2 *:*
- *Si* $\boxed{n=0}$ *alors* $L_\alpha = \emptyset$, $V_\alpha = \emptyset$ *et* $E_\alpha = \{\langle l_f, \emptyset, \emptyset, comHO[a][id_{A_2}]!, \emptyset, l'_f\rangle\}$
- *Si* $\boxed{n=1}$ *alors* $L_\alpha = \{Init\}$, $V_\alpha = \{y\}$ *et* $K_{L_\alpha}(Init) = c$
  et $E_\alpha = \{\langle l_f, \emptyset, \emptyset, comHO[a][id_{A_2}]!, \emptyset, Init\rangle, \langle Init, \emptyset, \emptyset, param[id_{A_2}][y]!, \emptyset, l'_f\rangle\}$
- *Si* $\boxed{n>1}$ *alors* $L_\alpha = \{Init_1,\ldots,Init_n\}$, $V_\alpha = \{y_1,\ldots,y_n\}$
  et $K_{L_\alpha}(Init_i) = c, i = 1,\ldots,n$
  et $E_\alpha = \{\langle l_f, \emptyset, \emptyset, comHO[a][id_{A_2}]!, \emptyset, Init_1\rangle,$
  $\langle Init_n, \emptyset, \emptyset, param[id_{A_2}][y_n]!, \emptyset, l'_f\rangle\} \cup E'_\alpha$
  où $E'_\alpha = \{e_1,\ldots,e_{n-1}\}$ *et* $e_i = \langle Init_i, \emptyset, \emptyset, param[id_{A_2}][y_{i+1}]!, \emptyset, Init_{i+1}\rangle$

La règle de transformation duale de la règle 3.12 est la règle de transformation 3.13. Celle-ci définit l'α-transformation d'une réception d'un terme d'ordre supérieur π-calcul. La règle 3.13 ajoute au modèle de TA auquel elle s'applique au moins une transition. Celle-ci est étiquetée par une action de réception appartenant au tableau *comHO* dans le but de recevoir l'identifiant du TA modélisant le terme d'ordre supérieur. À cette transition s'ajoutent n transitions (et n localités), chacune étiquetée par une action de réception utilisant le tableau *param*, correspondant aux n paramètres formels définis pour le terme d'ordre supérieur.

Règle 3.13 *[Transformation d'une réception d'ordre supérieur] Soient deux* π*-transformations telles que* $\mathcal{T}_\pi(A(x_1,\ldots,x_p)) = \mathcal{M}(id)$ *avec* $\mathcal{M}(id) = \langle L, l_0, E, V, C, Init, Inv, K_L\rangle$*. Si* l_f *représente l'état d'un agent* A *avant une action de réception à travers le canal* a *de l'agent* $B(y_1,\ldots,y_n)$*, modélisé par un automate ayant l'identifiant* id_B*, alors*

$$\mathcal{T}_\alpha(\,a(\,B(y_1,\ldots,y_n)\,)\ ,\ \mathcal{M}(id),\, l_f) = \Big(\mathcal{M}'(id), l'_f\Big)$$

$$\mathcal{M}' = \langle L \cup \{l'_f\} \cup L_\alpha, l_0, E \cup E_\alpha, V \cup \{id_B, a\} \cup V_\alpha, C, Init, Inv, K_{L_\alpha} \circ K_L\rangle$$

où $K_{L_\alpha}(l_f) = K_{L_\alpha}(l'_f) = o$ *et* n *l'arité de l'agent* B *:*
- *Si* $\boxed{n=0}$ *alors* $L_\alpha = \emptyset$, $V_\alpha = \emptyset$
  et $E_\alpha = \{\langle l_f, \{in\_id\}, \emptyset, comHO[a][in\_id]?, \{id_B := in\_id\}, l'_f\rangle\}$
- *Si* $\boxed{n=1}$ *alors* $L_\alpha = \{Init\}$, $V_\alpha = \{y\}$ *et* $K_{L_\alpha}(Init) = c$
  et $E = \{\langle l_f, \{in\_id\}, \emptyset, comHO[a][in\_id]?, \{id_B := in\_id\}, Init\rangle,$
  $\langle Init, \{in\}, \emptyset, param[id_B][in]?, \{y := in\}, l'_f\rangle\}$

– *Si* $\boxed{n > 1}$ *alors* $L_\alpha = \{Init_1, \ldots, Init_n\}$, $V_\alpha = \{y_1, \ldots, y_n\}$
et $K_{L_\alpha}(Init_i) = c, i = 1, \ldots, n$
et $E_\alpha = \{\langle l_f, \{in\_id\}, \emptyset, comHO[a][in\_id]?, \{id_B := in\_id, \}, Init_1\rangle,$
$\langle Init_n, \{in\}, \emptyset, param[id_B][in]!, \{y_n := in\}, l'_f\rangle\} \cup E'_\alpha$
où $E'_\alpha = \{e_1, \ldots, e_{n-1}\}$ *et* $e_i = \langle Init_i, \{in\}, \emptyset, param[id_B][in]!, \{y_i := in\}, Init_{i+1}\rangle$

La transformation d'un agent mobile en un TA ne fait pas apparaitre sa propriété « mobile ». La seule information qui communique entre l'émetteur et le récepteur de l'agent mobile est l'identifiant du TA représentant l'agent mobile. Le réseau d'automates de la figure 3.18 modélise les termes A, B et C (eq. 3.2) respectivement par les automates $\mathcal{M}_1$, $\mathcal{M}_2$ et $\mathcal{M}_3$.

Par exemple, l'émission d'un terme d'ordre supérieur, noté $C(w)$, à travers un canal z est modélisée par deux transitions (cf. Figure 3.18b) :

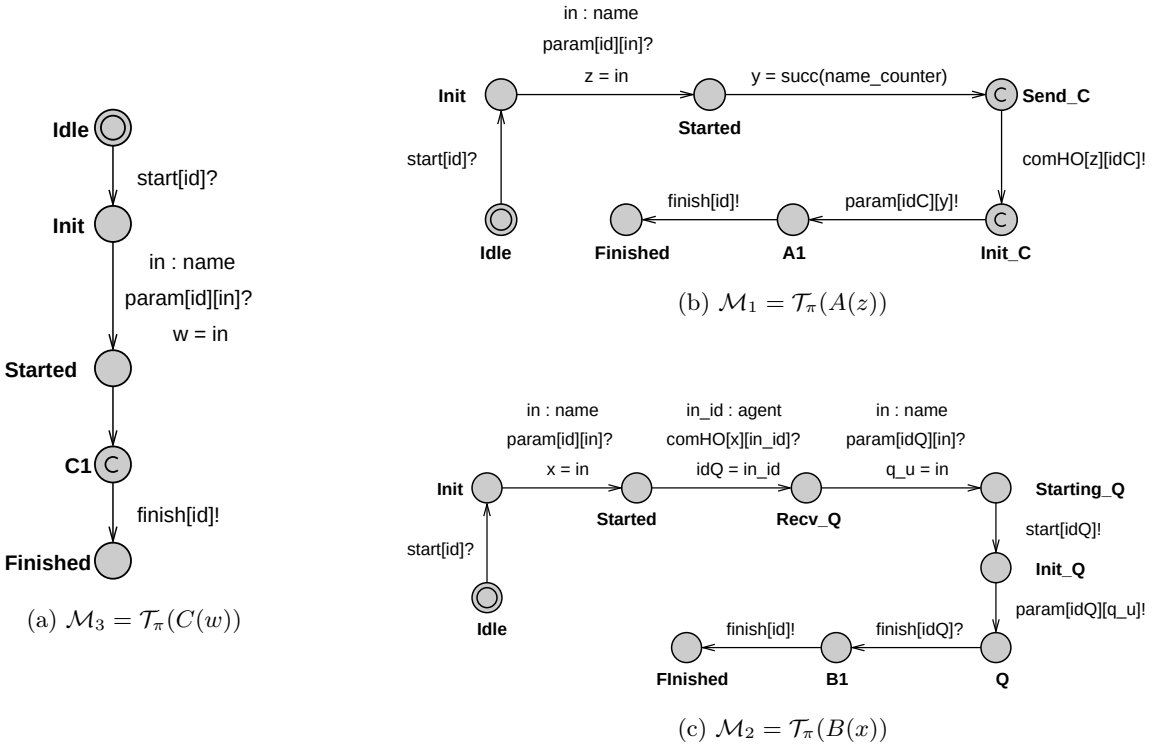
– la transition $\langle Send\_C, \emptyset, \emptyset, comHO[z][idC]!, \emptyset, Init\_C\rangle$ pour indiquer l'identifiant du *TA* correspond au terme d'ordre supérieur
– et la transition $\langle Init\_C, \emptyset, \emptyset, param[idC][y]!, \emptyset, A1\rangle$ pour envoyer la valeur de y pour le paramètre formel z défini par le terme C.

La réception de ce terme à travers le canal x est modélisée par deux transitions (cf. Figure 3.18c) : la transition $\langle Started, \{in\_id\}, \emptyset, comHO[x][in\_id]?, \{idQ = in\_id\}, Recv\_Q\rangle$ pour récupérer l'identifiant du TA et la transition $\langle Recv\_Q, \{in\}, \emptyset, param[idQ][in]?, \{q\_u = in\}, Starting\_Q\rangle$ pour récupérer la valeur du paramètre formel. L'exécution de l'agent mobile Q est alors modélisée grâce à une α-transformation suivant la règle 3.5.

3.2.2.7 Transformation de la récursion

Comme nous l'avons vu dans le chapitre 2, le π-calcul offre la possibilité d'écrire une définition récursive. Pour transformer ce type de comportement dans un réseau d'automates, nous donnons la règle 3.14. Celle-ci définit l'α-transformation d'un appel récursif direct. Nous appelons un appel récursif « direct » une récursion obtenue à partir d'un seul agent. Par exemple, l'équation 3.3 présente une récursion directe définie par un unique terme A et l'équation 3.4 une récursion « normale » obtenue à l'aide de deux termes, les termes B et C. Notons que la transformation de la récursion définie par l'équation 3.4 est possible en appliquant au préalable une substitution de C dans B. Néanmoins, cette démarche ne permettrait pas de garder un parallèle entre les agents π-calcul et les automates générés. Il serait alors difficile de pouvoir écrire des propriétés qui auraient un sens vis-à-vis de la spécification réalisée en π-calcul.

$$A(\,x\,) \stackrel{def}{=} (\,\nu\,u\,)\,\Big(\overline{x}\,\langle\,u\,\rangle\,.\,\tau\,.\,u(\,b\,)\,.\,A(\,b\,)\,\Big) \tag{3.3}$$



(a) $\mathcal{M}_3 = \mathcal{T}_\pi(C(w))$

(b) $\mathcal{M}_1 = \mathcal{T}_\pi(A(z))$

(c) $\mathcal{M}_2 = \mathcal{T}_\pi(B(x))$

FIGURE 3.18 – Modélisation d'une communication d'ordre supérieur. À partir de l'équation 3.2 , la transformation de la communication d'un agent défini par le terme C, modélisé par $\mathcal{M}_3$ (a) entre les agents A et B, modélisés respectivement par $\mathcal{M}_1$ (b) et $\mathcal{M}_2$ (c)).

$$B(\,x\,) \stackrel{def}{=} (\,\nu\, u\,)\left(\overline{x}\,\langle\, u\,\rangle\,.\,\tau\,.\,C(\,u\,)\,\right) \text{ et } C(\,z\,) \stackrel{def}{=} \overline{z}\,\langle\, w\,\rangle\,.\,\tau\,.\,B(\,w\,) \tag{3.4}$$

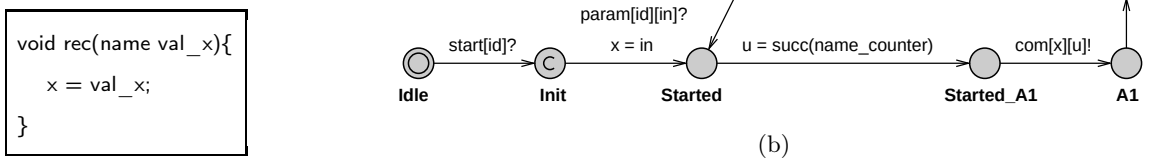
La règle 3.14 ajoute une seule transition au modèle de TA sur lequel elle est appliquée. Cette transition permet d'affecter une nouvelle valeur à chaque variable locale appartenant au modèle de TA et modélisant les paramètres formels de l'agent π-calcul qu'il représente. L'affectation est réalisée par la fonction *rec* qui est définie localement à chaque modèle d'automate. Cette fonction prend autant de paramètres que l'agent π-calcul a de paramètres formels et assigne à chaque variable locale modélisant un paramètre formel la valeur fournie en paramètre de la fonction *rec*. Par exemple, l'agent A (eq. 3.3) définit un paramètre formel alors la fonction *rec* locale au modèle d'automate présenté dans la figure 3.19 définit un seul paramètre.

Règle 3.14 *[Transformation d'une récursion directe] Soit $A(x_1, \ldots, x_n)$ la définition d'un agent π-calcul et $\mathcal{M}(id)$ un modèle d'automate tel que $\mathcal{T}_\pi(A) = \mathcal{M}(id)$*
avec $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L\rangle$. Si l_f modélise l'état d'un agent A avant un appel récursif direct de A alors

$$\mathcal{T}_\alpha^{rec}(\,A(y_1, \ldots, y_n),\, \mathcal{M}(id),\, l_f) = \Big(\langle L, l_0, E \cup E_\alpha, C, Affect, Inv, K_{L_\alpha} \circ K_L\rangle, Started\Big)$$

où $E_\alpha = \{\langle l_f, \emptyset, \emptyset, \emptyset, \{rec(y_1, \ldots, y_n)\}, Started\rangle\}$, $K_{L_\alpha}(l_f) = c$ et $K_{L_\alpha}(Started) = o$.

Nous appliquons la règle 3.14 dans la transformation du terme défini par l'équation 3.3. Nous pouvons observer que la récursion est modélisée par la transition définie par le tuple $\langle A3, \emptyset, \emptyset, \emptyset, \{rec(b)\}, Started\rangle$ ce qui permet d'affecter la valeur de la variable b au paramètre formel x.



(a) (b)

FIGURE 3.19 – Modélisation d'une récursion avec le modèle d'automate résultat de la transformation $\mathcal{T}_\pi$ de la définition d'agent A (eq. 3.3)

3.2.2.8 Transformation d'une somme

La dernière règle de transformation que nous présentons dans cette section est celle appliquée au comportement d'un agent défini par une somme. Celle-ci, dont la notation π-calcul est l'opérateur $+$, représente un choix dans l'évolution d'un agent. Nous pouvons alors écrire la forme générale de la somme de la manière suivante

$$P = \sum_{i=1}^{n} \alpha_i \,.\, P_i$$

où pour chaque état P, dans lequel un choix se pose, n transitions étiquetées $\alpha_1, \ldots, \alpha_n$ mènent respectivement aux états $P_1, \ldots, P_n$.

La règle 3.15 définit qu'une somme π-calcul se transforme vers un modèle de TA à l'aide de la composition des deux α-transformations de ses termes. Chacune des α-transformations indique la même localité comme troisième paramètre : la localité correspondant à l'état de l'agent avant l'évaluation de la somme. Il est alors possible d'écrire le modèle d'automate $\mathcal{M}'$ comme le résultat de l'α-transformation d'une somme de n termes vers le modèle de TA $\mathcal{M}$ de la manière suivante (avec la localité l_f représentant le système avant l'évaluation de la somme) :

$$\Big(\mathcal{T}_\alpha(\,\alpha_n \,.\, P_n) \circ \cdots \circ \mathcal{T}_\alpha(\,\alpha_2 \,.\, P_2) \circ \mathcal{T}_\alpha(\,\alpha_1 \,.\, P_1)\Big)\Big(\mathcal{M}(id),\, l_f\Big)$$

Règle 3.15 *[Transformation d'une somme] Soit $A(x_1, \ldots, x_n)$ la définition d'un agent π-calcul et $\mathcal{M}$ un modèle d'automate tel que $\mathcal{T}_\pi(A) = \mathcal{M}(id)$ avec $\mathcal{M}(id) = \langle L, l_0, E, V, C, Assign, Inv, K_L\rangle$.*

Si l_f modélise l'état d'un agent A avant le choix entre les processus P et Q, noté $P + Q$, alors la transformation de cette somme est la composée de deux α-transformations telles que

$$\mathcal{T}_\alpha(P + Q, \mathcal{M}(id), l_f) = \Big(\mathcal{T}_\alpha(Q) \circ \mathcal{T}_\alpha(P)\Big)\Big(\mathcal{M}(id), l_f\Big)$$

Par exemple, le système π-calcul défini par $System \stackrel{def}{=} (\nu\, x)\ \Big(A(\,x\,) \mid B(\,x\,)\ \Big)$

$$\text{avec } \begin{cases} A(\,x\,) \stackrel{def}{=} (\nu\, u, w)\ \Big(\overline{x}\langle\, u, w\,\rangle\ .\ \Big(u(\,a\,)\ .\ R + w(\,a\,)\ .\ S\ \Big)\Big) \\ B(\,z\,) \stackrel{def}{=} z(\,b, d\,)\ .\ \Big((\nu\, c)\ \overline{b}\langle\, c\,\rangle\ .\ 0 + (\nu\, e)\ \overline{d}\langle\, e\,\rangle\ .\ B(\,d\,)\ \Big) \end{cases} \tag{3.5}$$

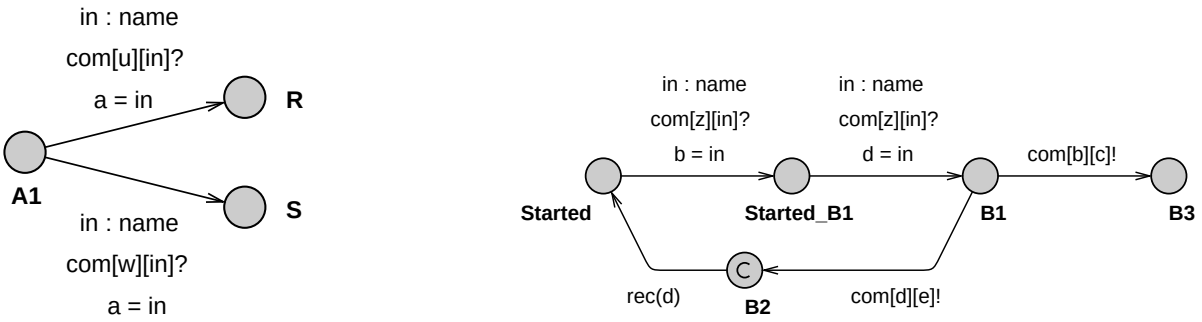
Ce système suit la réduction suivante

$$System \xrightarrow{\tau} \Big(u(\,a\,)\ .\ R + w(\,a\,)\ .\ S\ \Big) \mid \Big((\nu\, c)\ \overline{u}\langle\, c\,\rangle\ .\ 0 + (\nu\, e)\ \overline{w}\langle\, e\,\rangle\ .\ B(\,d\,)\ \Big)$$

La somme offre alors deux possibilités pour la réduction du système $System$:

– Soit les deux agents communiquent par le canal u et le système évolue en $R \mid 0$
– Soit les deux agents communiquent par le canal w et le système évolue en $S \mid B(\,d\,)$

La figure 3.20 présente les états et transitions ajoutés aux modèles d'automate résultat des transformations des sommes présentes dans les définitions des agents A et B de l'équation 3.5.

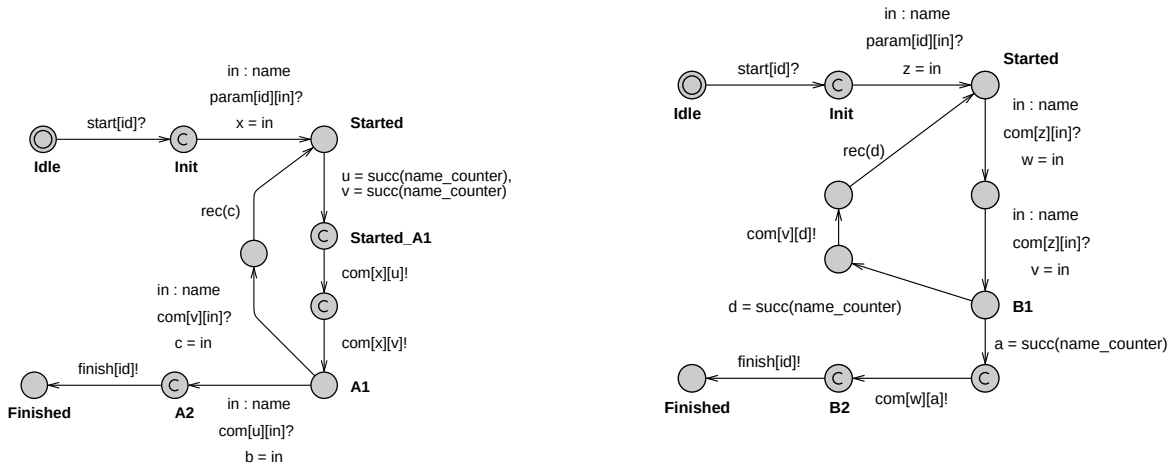


(a) $\mathcal{T}_\alpha(u(\,a\,) + w(\,a\,)\ , \mathcal{M}_1(id), A1)$

(b) $\mathcal{T}_\alpha(\overline{b}\langle\, c\,\rangle + \overline{d}\langle\, e\,\rangle\ , \mathcal{M}_2(id), B1)$

FIGURE 3.20 – Modélisation d'une somme dans un modèle d'automate avec les états et les transitions ajoutées lors des α-transformations appliquées aux modèles d'automate $\mathcal{M}_1$ et $\mathcal{M}_2$ tels que $\mathcal{T}_\pi(A(x)) = \mathcal{M}_1$ et $\mathcal{T}_\pi(B(z)) = \mathcal{M}_2$

Nous pouvons maintenant appliquer des π-transformations sur les définitions des agents A et B de l'équation 3.1. La figure 3.21 présente les modèles de TA obtenus par ces π-transformations.



(a) $\mathcal{M}_1$: Résultat de $\mathcal{T}_\pi(A(x))$

(b) $\mathcal{M}_2$: Résultat de $\mathcal{T}_\pi(B(z))$

FIGURE 3.21 – Modélisation du résultat des π-transformations des définitions des agents A et B de l'équation 3.1.

3.3 Vérification d'une architecture logicielle

Dans cette section, nous appliquons les règles de transformation, définies dans la section 3.2, à la spécification $MCA\pi Spec$ écrite dans le chapitre 2. Le réseau d'automates temporisés UPPAAL, résultat de cette transformation, permet d'énoncer des propriétés afin de vérifier cette architecture logicielle pour le calcul parallèle.

3.3.1 Transformation d'une spécification π-calcul

En appliquant l'opérateur de transformation 3.14 sur la spécification $MCA\pi Spec$, nous obtenons un réseau d'automates temporisés dans lequel chaque automate est une instance d'un modèle d'automate. Pour rendre plus simple la lecture du réseau d'automates temporisés résultant de la transformation $\mathcal{T}_S(MCA\pi Spec)$, nous choisissons de réduire le nombre de termes qui composent la spécification $MCA\pi Spec$. En effet, la transformation de tous les termes définis dans $MCA\pi Spec$ rendrait la lecture de la suite de ce chapitre moins agréable car les figures présentées seraient alors beaucoup plus complexes et, de ce fait, moins lisibles. Dans ce but, nous supprimons les termes $Property$ (eq. 2.48) et $PropertyDirectory$ (eq. 2.49) modélisant les propriétés d'un cas de calcul et les termes $DataHandler$ (eq. 2.39) et $DHDirectory$ (eq. 2.40) modélisant une mémoire distribuée. La suppression de ces termes entraine la modification de certains termes, en particulier ceux qui modélisent un cas de calcul. Nous proposons donc une nouvelle définition des termes $ComputationCase$ (eq. 3.6), $Master$ (eq. 3.7), $TaskHandler$ (eq. 3.8), $Task$ (eq. 3.10) et $ComputeAgent$ (eq. 3.9). Dans cette section nous présentons le résultat de la transformation $\mathcal{T}_S(MCA\pi Spec)$ en trois parties : la déclaration du système UPPAAL, les automates qui modélisent notre espace de travail, les processus $CaseDirectory$ (eq. 2.61) et $Worker$ (eq. 2.64), et

enfin les automates qui modélisent les agents spécifiques à un cas de calcul.

3.3.1.1 Déclaration du système UppAal

La sémantique des automates temporisées ne permet pas d'ajouter dynamiquement des automates (c'est à dire au cours de l'exécution du système). Vouloir modéliser le coté adaptatif de notre architecture logicielle ne consiste donc pas à ajouter ou supprimer des TA au système UppAal au cours d'une exécution du système. Il est indispensable de connaître le nombre total de processus $Worker$ et de cas de calcul qui seront présents au cours d'une exécution du système avant le début de l'exécution de celui-ci. Par exemple, l'ajout d'un processus $Worker$ au système se fait par une action de réception sur le canal $start[id]$ du TA qui représente le processus $Worker$ à ajouter. La figure 3.22 présente les déclarations globales au système UppAal auxquelles il faut ajouter la déclaration de canaux et de la variable *name_counter* communs à toutes les transformations suivant la définition 3.14. Les constantes `WORKERS` et `CC` permettent de définir respectivement le nombre de processus travailleurs et le nombre de cas de calcul présents dans le système que nous modélisons.

Pour chaque modèle d'automate résultant de la π-transformation d'un terme appartenant à la spécification $MCA\pi Spec$, nous définissons un nouveau type UppAal (cf. Section 3.1.4). Chaque type permet alors de borner les identifiants (ceux-ci sont définis par le paramètre *id* de chaque modèle) des instances d'automate existant pour chaque modèle d'automate (cf. Définition 3.13). Cette solution facilite grandement la définition du système UppAal (cf. Figure 3.23). En effet, pour définir le système, nous indiquons uniquement le nom du modèle d'automate et le nombre d'instance de ce modèle dans le système est alors égale au nombre de valeurs possibles définies par le type de son identifiant. Par exemple, le type `id_w` définit deux entiers : les entiers `2` et `3` (`int[2, 2 + WORKERS-1]`) . De plus, notons les déclarations permettant la gestion des pannes de $Worker$:

- un tableau de canaux de diffusion `cc_finish` utilisé par le modèle `CaseHandler` (cf. Figure 3.32) pour prévenir les $Workers$ de la fin d'un cas de calcul,
- des tableaux de canaux `stopWorker` et `stopCA` permettant de modéliser l'arrêt d'un agent $Worker$ pendant que celui-ci traite une tâche de calcul. Ces canaux sont utilisés dans les modèles `Worker` (cf. Figure 3.25) et `ComputeAgent` (cf. Figure 3.34a),
- une constante `maxCATimeExec` définissant la durée maximum que peut prendre l'exécution d'un code de calcul défini pour un *ComputeAgent*. Cette constante est utilisée dans les modèles `ComputeAgent` (cf. Figure 3.34a) et `Task` (cf. Figure 3.33)

Dans la suite de cette section, nous présentons les modèles d'automates utilisés dans la déclaration du système UppAal. Chacun de ces modèles déclare un paramètre `id` typé par le type correspondant défini dans la figure 3.22. Par exemple, pour le modèle d'automate `Worker` (cf. Figure 3.25), résultat de la transformation $\mathcal{T}_\pi(Worker)$, nous déclarons le type `id_w`. Celui-ci fixe les valeurs possibles pour les variables de ce type à 2 et 3 (`int[2,3]`). Ainsi le système UppAal contient deux automates modélisant des processus $Worker$: les automates `Worker(2)` et `Worker(3)`.

```
1  // Configuration
2  const int WORKERS = 2; // nombre de processus travailleurs
3  const int CC = 1; // nombre de cas de calcul
4  const int TASKS_PER_CC = 2; // nombre de tâches par cas de calcul
5
6  // Identifiants
7  typedef int[1, 1 + WORKERS + 1 + CC*4 + CC*TASKS_PER_CC*3] agent;
8  const agent idCD = 1; // identifiant du processus CaseDirectory
9  typedef int[idCD,idCD] id_cd;
10 typedef int[2, 2 + WORKERS-1] id_w; // type pour les identifiants des processus Worker
11 const agent idUser1 = 2 + WORKERS; // type pour l'identifiant du processus User1
12 typedef int[idUser1,idUser1] id_user;
13 const agent idCC1 = 3 + WORKERS; // type pour l'identifiant du processus ComputationCase utilisé par User1
14 typedef int[idCC1,idCC1+CC-1] id_cc; // type pour les identifiants des processus ComputationCase
15 typedef int[idCC1+CC , idCC1+CC*2-1] id_ch; // type pour les identifiants des processus CaseHandler
16 typedef int[idCC1+CC*2, idCC1+CC*3-1] id_m; // type pour les identifiants des processus Master
17 typedef int[idCC1+CC*3, idCC1+CC*4-1] id_td; // type pour les identifiants des processus TaskDirectory
18 typedef int[idCC1+CC*4, idCC1+CC*4+CC*TASKS_PER_CC-1] id_th; // type pour les TaskHandler
19 typedef int[idCC1+CC*4+CC*TASKS_PER_CC, idCC1+CC*4+CC*TASKS_PER_CC*2 -1] id_t; // type pour les Task
20 typedef int[idCC1+CC*4+CC*TASKS_PER_CC*2, idCC1+CC*4+CC*TASKS_PER_CC*3 -1] id_ca; // type pour les CA
21
22 // Gestion des pannes de Worker
23 broadcast chan cc_finish[name];
24 chan stopWorker[id_w];
25 chan stopCA[id_ca];
26 const int maxCATimeExec = 100;
```

FIGURE 3.22 – Déclarations globales du système UPPAAL.

```
1  system System, CaseDirectory, User, Worker,
2    ComputationCase, Master, TaskHandler, Task,
3    ComputeAgent, CaseHandler, TaskDirectory;
```

FIGURE 3.23 – Déclaration du système UPPAAL.

3.3.1.2 Transformation de l' « espace de travail »

L'équation 2.70 à la fin du chapitre 2 définit un espace de travail comme étant un système composé d'un terme $CaseDirectory$ (eq. 2.61), modélisant un annuaire de cas de calcul, et un terme $Worker^m$, modélisant une ferme de m travailleurs. La figure 3.24 présente le modèle d'automate `CaseDirectory` résultat de la π-transformation $\mathcal{T}_\pi(CaseDirectory)$.

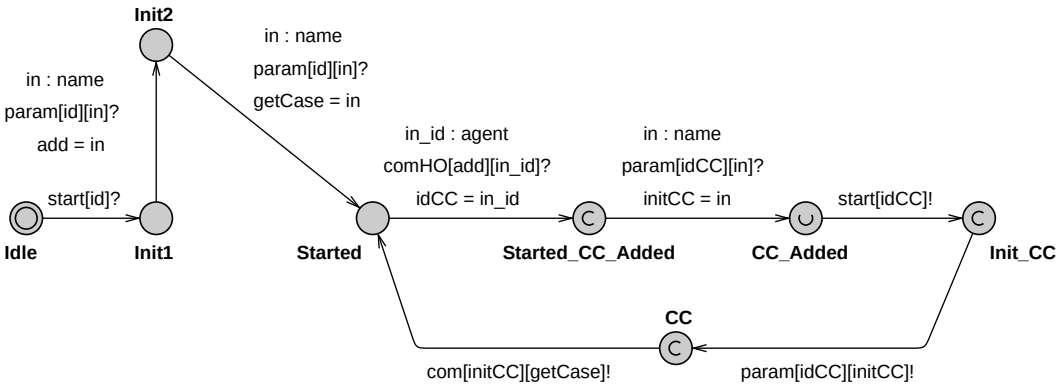


FIGURE 3.24 – Modèle d'automate `CaseDirectory` : résultat de la π-transformation du terme $CaseDirectory$ (eq. 2.61)

Le terme $Worker^m$ modélise une ferme de m travailleurs dans laquelle le terme $Worker$ (eq. 2.64) définit un processus travailleur capable de traiter toutes les tâches de tous les cas de calcul disponibles dans l'annuaire modélisé par le terme $CaseDirectory$ (cf. Section 2.2.4.2). La figure 3.25 propose le modèle d'automate `Worker`, résultat de la transformation $\mathcal{T}_\pi(Worker)$. Les π-transformations des deux termes $Worker_{started}$ et $Worker_{connected}$ étant réunis dans le même modèle d'automate. Notons la présence de la localité `On_Error` qui modélise l'arrêt d'un $Worker$ à la suite d'une panne. Le passage dans cet état se concrétise par une transition étiquetée par l'action `stopCA[idCA]!` qui représente l'arrêt du code de calcul en cours de traitement au moment de la panne. Nous modélisons aussi l'arrêt volontaire d'un terme $Worker$. Pour cela, nous ajoutons la transition étiquetée par l'action `stopWorker[id]?` suivie de la transition étiquetée par l'action `stopCA[idCA]!`. Ces deux actions doivent être exécutées de manière atomique, c'est pourquoi nous définissons la localité `Worker_Stop` comme *committed*. L'incidence d'une panne d'un $Worker$ sur l'évaluation d'un cas de calcul est vérifiée dans la section 3.3.2.1 alors que celle de l'arrêt volontaire d'un terme $Worker$ est vérifiée dans la section 3.3.2.2.

Le nombre de travailleurs qui composent la ferme est défini par la constante `WORKERS` (cf. Figure 3.22) et le paramètre *id* du modèle d'automate `Worker` est de type `id_w` (cf. Figure 3.23), ce qui signifie qu'il y a autant d'instance de ce modèle que d'entiers définis par le type `idw`.

3.3.1.3 Transformation d'un cas de calcul

Pour simplifier la lecture du résultat de la transformation de la spécification $MCA\pi Spec$, nous supprimons les termes modélisant les propriétés et l'accès aux données d'un cas de calcul. Nous simplifions alors le terme $ComputationCase$ (eq. 2.58) en lui retirant les agents $DHDirectory$ et $PropertyDirectory$, ce qui donne le nouveau terme $ComputationCase$ de l'équation 3.6.

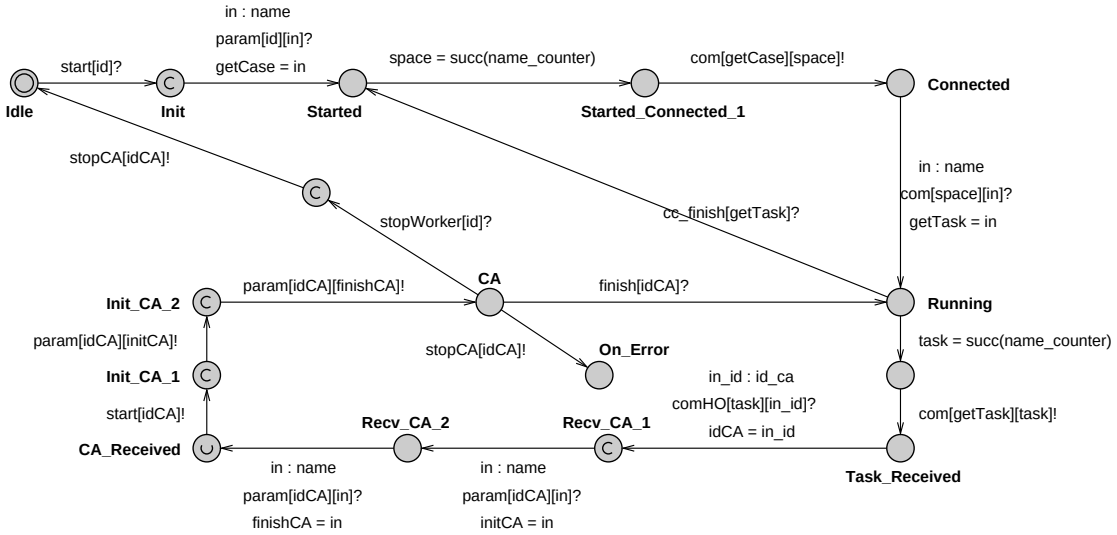


FIGURE 3.25 – Modèle d'automate `Worker` : résultat de la π-transformation du terme $Worker$ (eq. 2.64)

$$
\begin{aligned}
ComputationCase(\, init \,) \stackrel{def}{=}\ & init(\, getCase \,) \\
& . \left(\begin{array}{ll} & add_T, \\ \nu & getTask, \\ & finish \end{array} \right) \left(\begin{array}{ll} & Master(\, getTask, add_T, finish \,) \\ \mid & TaskDirectory(\, add_T \,) \\ \mid & CaseHandler(\, getCase, getTask, finish \,) \end{array} \right)
\end{aligned} \tag{3.6}
$$

La figure 3.26 présente le résultat de la π-transformation du terme $ComputationCase$ (eq. 3.6). La règle 3.4 est utilisée à trois reprises en utilisant les identifiants définis dans la déclaration locale au modèle d'automate `ComputationCase` (cf. Figure 3.28a) :

- `idM` pour l'instance du modèle d'automate `Master` (cf. Figure 3.27) modélisant le processus maître du cas de calcul ;
- `idTD` pour l'instance du modèle d'automate `TaskDirectory` (cf. Figure 3.30) modélisant l'annuaire de tâches ;
- `idCH` pour l'instance du modèle d'automate `CaseHandler` (cf. Figure 3.32) modélisant l'agent $CaseHandler$ qui a la gestion des processus travailleurs participant au cas de calcul.

Lorsqu'une instance du modèle `ComputationCase` se situe dans la localité `M`, cela signifie que les processus $Master$, $TaskDirectory$ et $CaseHandler$ sont démarrés. Nous présentons dans la suite de cette section le résultat des π-transformations de ces trois processus π-calcul.

Nous enlevons aux termes $Master$ (eq. 2.60) et $TaskHandler$ (eq. 2.45) leur paramètre formel add_{DH}. Celui-ci est utilisé pour communiquer avec un processus $DHDirectory$ mais ce processus ne fait pas partie du système que nous transformons. Nous obtenons alors les termes $Master$

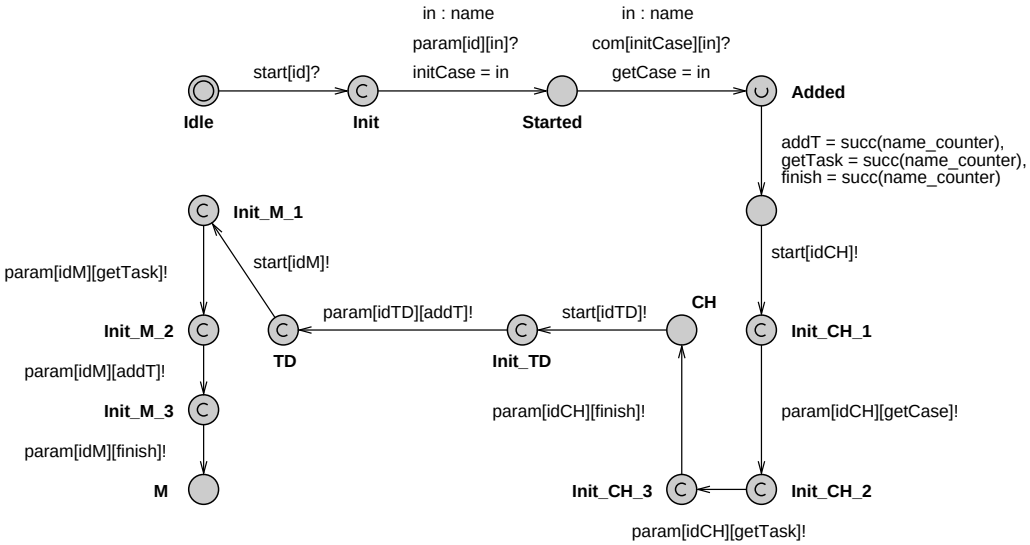


FIGURE 3.26 – Modèle d'automate `ComputationCase` : résultat de la π-transformation du terme $ComputationCase$ (eq. 3.6)

(eq. 3.7) et $TaskHandler$ (eq. 3.8).

$$Master(\, getTask, add_T, finish\,) \stackrel{def}{=} \left(\begin{array}{c} TaskHandler_1(\, getTask, add_T\,) \\ \quad \mid \cdots \mid TaskHandler_n(\, getTask, add_T\,) \end{array} \right) . \tau . \overline{finish} \tag{3.7}$$

avec

$$TaskHandler(\, getTask, add_T\,) \stackrel{def}{=} (\, \nu\, init, finish\,)\, \overline{add_T}\, \langle\, Task\,\rangle\, .\, \overline{init}\, .\, finish \tag{3.8}$$

La π-transformation du terme $Master$ (eq. 3.7) donne le modèle d'automate `Master` (cf. Figure 3.27). Nous pouvons remarquer la déclaration de la fonction $startTH$ (cf. Figure 3.28b) utilisée lors de la transaction entre les états `Start_TH_2` et `Started`. Celle-ci permet de connaître la prochaine instance de `TaskHandler` à démarrer à l'aide de l'action de synchronisation `start[idTH]!`. Le nombre d'agent $TaskHandler$ à démarrer est défini par la constante `TASKS_PER_CC` (cf. Figure 3.22) car ce nombre correspond aux nombres de tâches par cas de calcul.

Nous obtenons le modèle d'automate `TaskHandler` (cf. Figure 3.29) par la π-transformation du terme $TaskHandler$ (eq. 3.8). Son rôle est d'ajouter un agent $Task$ à l'annuaire $TaskDirectory$. L'agent $Task$ est représenté par un entier `idT` de type `id_t` (cf. Figure 3.22). La valeur de `idT` est égale à `id + TASKS_PER_CC` avec `id` l'identifiant de l'instance du modèle d'automate `TaskHandler` correspondant à cette tâche.

La modélisation dans UPPAAL d'un annuaire de tâches est réalisée par la π-transformation du terme $TaskDirectory$ (eq. 2.20).

La figure 3.31 propose la représentation d'une communication d'ordre supérieur à l'aide de l'outil

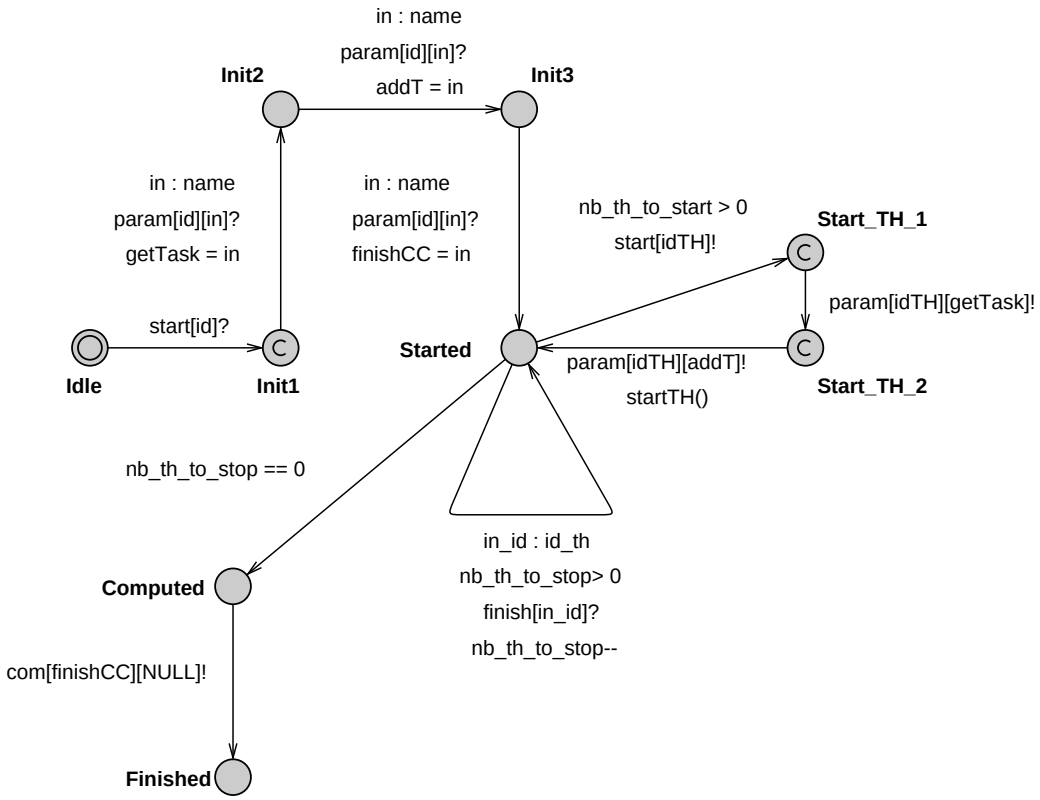


FIGURE 3.27 – Modèle d'automate `Master` : résultat de la π-transformation du terme $Master$ (eq. 3.7)

```
1  agent idM = id + CC∗2;
2  agent idCH = id + CC;
3  agent idTD = id + CC∗3;
```

(a) `ComputationCase`

```
1  int nb_th_to_start = TASKS_PER_CC;
2  int nb_th_to_stop = TASKS_PER_CC;
3  id_th idTH = id+CC∗2;
4
5  void startTH(){
6      if(−−nb_th_to_start > 0){
7          idTH++;
8      }
9  }
```

(b) `Master`

FIGURE 3.28 – Déclarations locales des modèles d'automates `ComputationCase` et `Master`.

UPPAAL. Il s'agit des états et des transitions ajoutés par les deux α-transformations suivantes :
– $\mathcal{T}_\alpha(\overline{add_T}\,\langle Task\rangle\,, TaskHandler, Started)$ (cf. Règle 3.12) pour l'émission d'un agent $Task$;
– $\mathcal{T}_\alpha(add(Task)\,, TaskDirectory, Started)$ (cf. Règle 3.13) pour la réception d'un agent $Task$.

La π-transformation du terme $CaseHandler$ (eq. 2.59) est plus complexe que les autres transformations présentées jusqu'à maintenant. Ceci est dû à la modélisation par le terme π-calcul

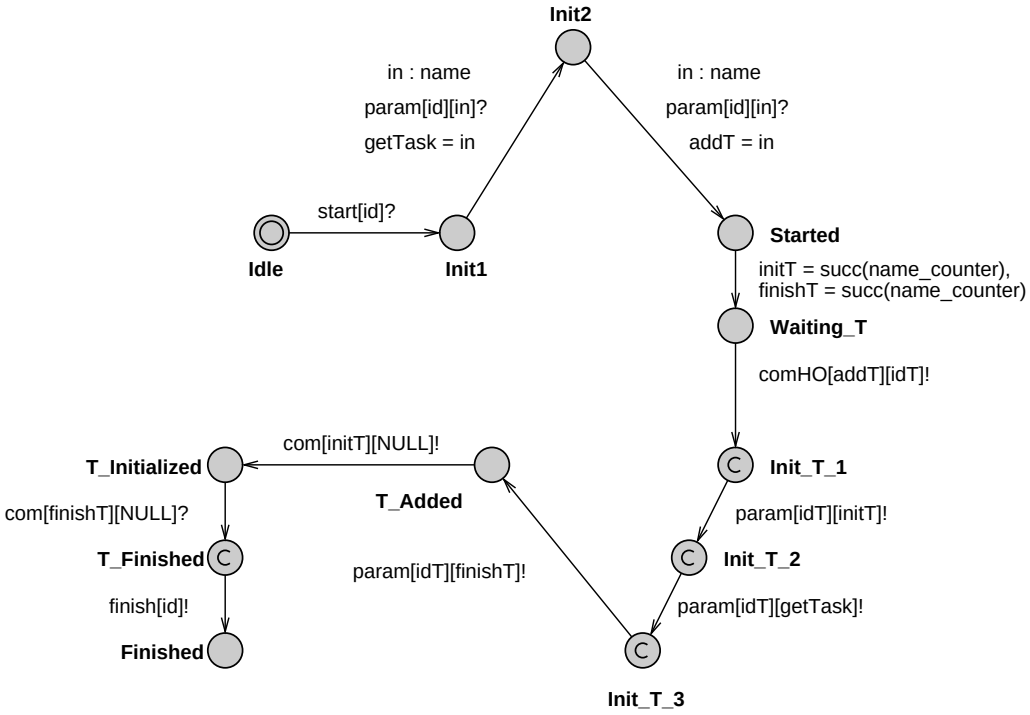


FIGURE 3.29 – Modèle d'automate `TaskHandler` : résultat de la π-transformation du terme $TaskHandler$ (eq. 3.8)

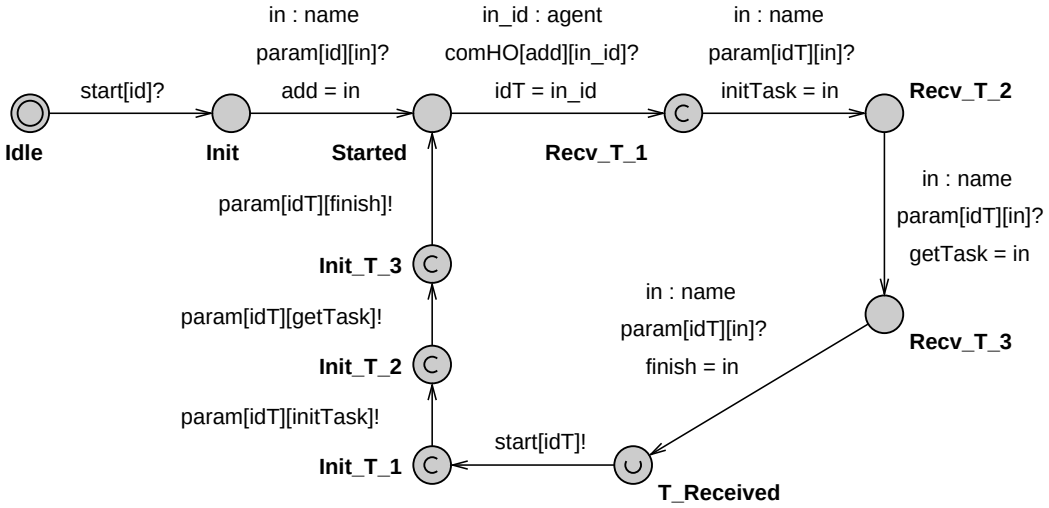
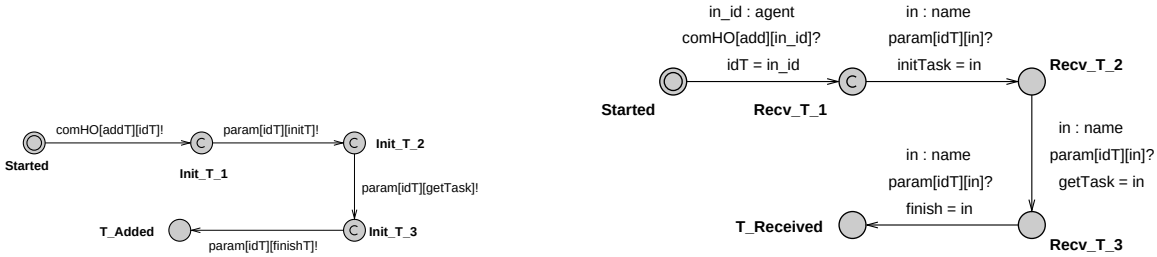


FIGURE 3.30 – Modèle d'automate `TaskDirectory` : résultat de la π-transformation du terme $TaskDirectory$ (eq. 2.20)

de la fin d'un cas de calcul. Après une réception sur son canal $finish$, un agent $CaseHandler$ doit prévenir tous les agents $Worker$ qui participent au cas de calcul de la fin de celui-ci. Cette action est présentée par l'équation 2.69. Pour représenter cette action dans le modèle d'automate `CaseHandler` (cf. Figure 3.32) , nous utilisons le tableau `cc_finish` déclaré dans la figure 3.22. Celui-ci permet de définir l'action de synchronisation `cc_finish[getTask]!` ce qui n'impose pas la



(a) Émission d'une tâche

(b) Réception d'une tâche

FIGURE 3.31 – Transformation de la communication entre un processus $TaskHandler_i$ et un processus $TaskDirectory$ lors de l'ajout d'une tâche $Task_i$ à l'annuaire de tâches.(a) Envoi d'un agent $Task_i$ par le processus $TaskHandler$ à travers le canal add_T. (b) Réception d'un agent $Task_i$ par le processus $TaskDirectory$ à travers le canal add

présence d'une action de réception duale qui serait définie par un automate suivant le modèle `Worker` (cf. Figure 3.25). De cette manière, l'arrêt d'un terme $Worker$ au cours de l'exécution d'un cas de calcul ne bloque pas l'exécution du $CaseHandler$ correspondant.

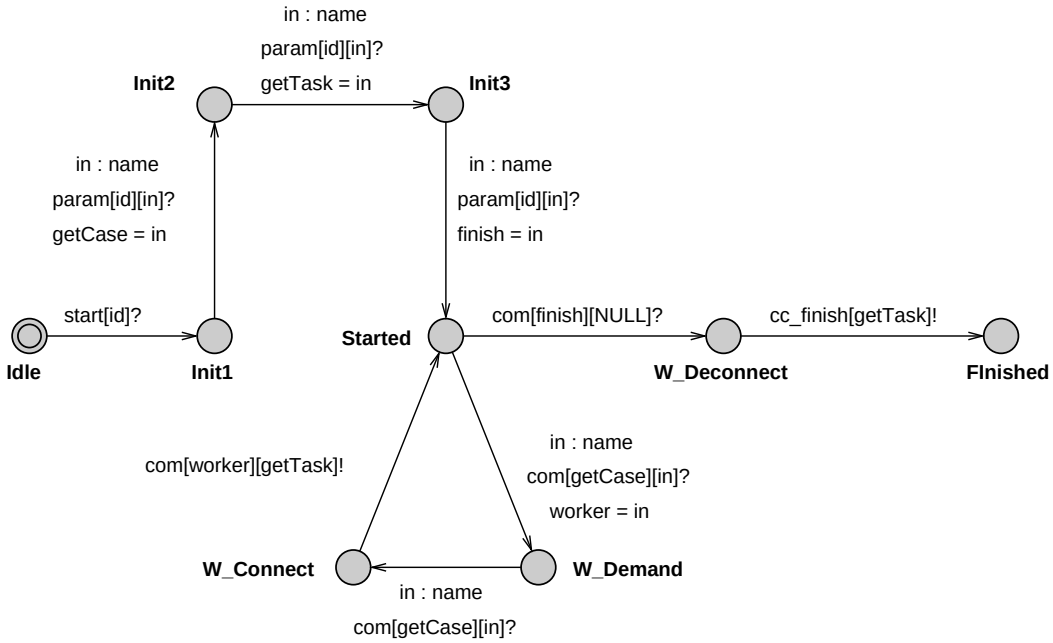


FIGURE 3.32 – Modèle d'automate `CaseHandler` : résultat de la π-transformation du terme $CaseHandler$ (eq. 2.59)

La modélisation en π-calcul de l'agent mobile contenant le code de calcul de chaque tâche est donné par le terme $ComputeAgent$ (eq. 3.6). Ce terme définit deux paramètres formels $init$ et $finish$ avec une action de réception sur le canal $init$, une action interne et une action d'émission sur le canal $finish$.

$$ComputeAgent(\,init, finish\,) \stackrel{def}{=} init\ .\ \tau\ .\ \overline{finish}\ .\ 0 \tag{3.9}$$

L'action d'émission sur le canal *init* et l'action de réception sur le canal $finish$ sont présentes dans le terme $Task$ (eq. 3.10) représente une tâche du cas de calcul modélisé par le terme $ComputationCase$ (eq. 3.6).

$$\begin{aligned} &Task(\,init, getTask, finish\,) \stackrel{def}{=} \\ &\quad init\ .\ getTask(\,worker\,) \\ &\quad .\ \left(\begin{array}{cc} \nu & initCA, \\ & finishCA \end{array}\right) \left(\begin{array}{c} \overline{worker}\,\langle\, CA(initCA, finishCA\,\rangle \\ .\ \overline{initCA}\ .\ finishCA(\,result\,) \end{array}\right) \\ &\quad .\ \overline{finish}\ .\ 0 \end{aligned} \tag{3.10}$$

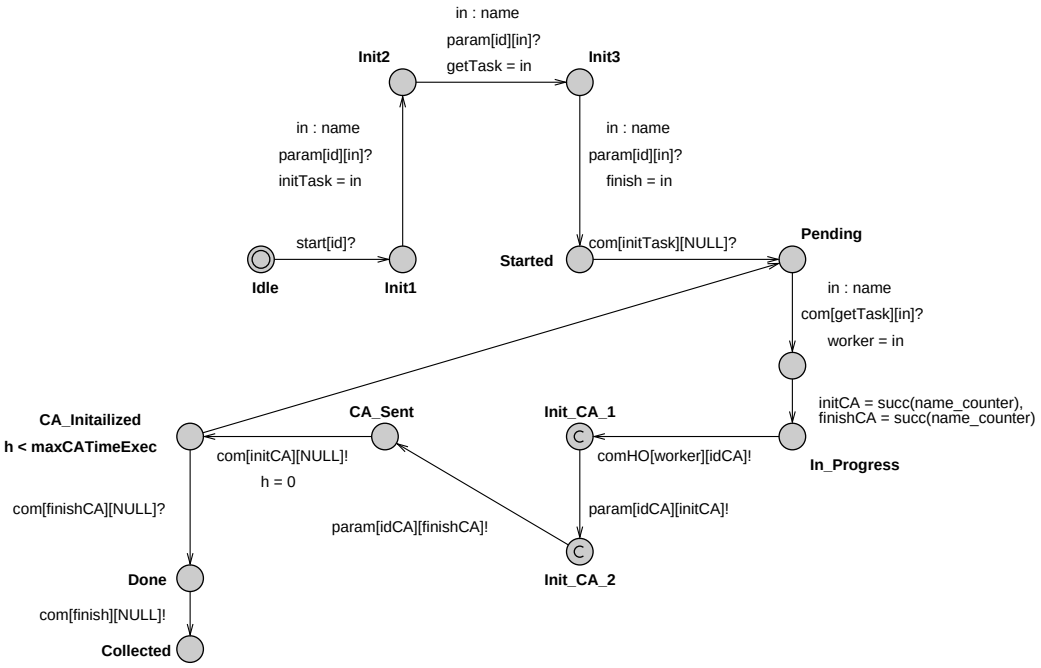
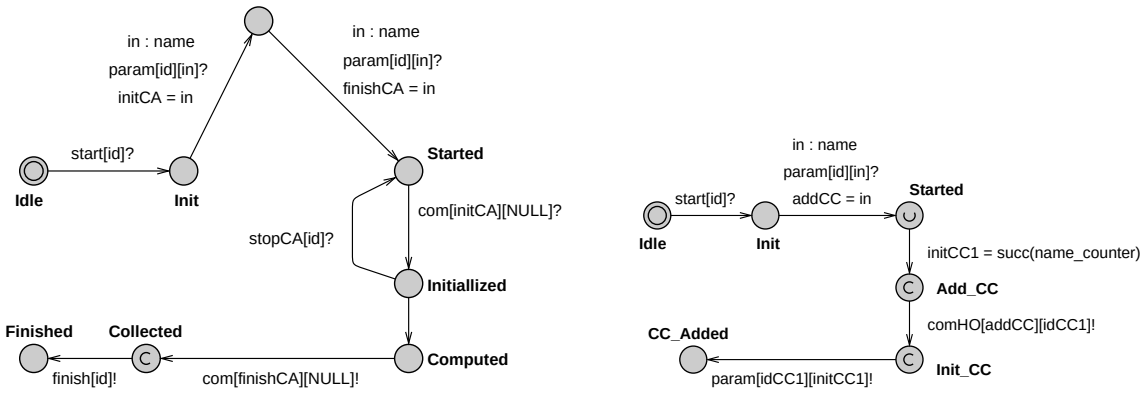


FIGURE 3.33 – Modèle d'automate `Task` : résultat de la π-transformations du terme principal Tas (eq. 3.10)

La π-transformation du terme *ComputeAgent* (eq. 3.6) donne le modèle d'automate `ComputeAgent` (cf. Figure 3.26) et la π-transformation du terme $Task$ (eq. 3.10) donne le modèle d'automate `Task` (cf. Figure 3.33). Notons l'ajout d'une transition dans chacun de ces deux modèles afin de pouvoir prendre en compte l'arrêt spontané d'un terme $Worker$ pendant le traitement d'une tâche. Dans le modèle `ComputeAgent` nous ajoutons la transition étiquetée par l'action `stopCA[id]?`. Cette action est l'action duale de celle définie dans le modèle `Worker` (cf. Figure 3.25). Il est alors important de signaler que l'arrêt d'un *ComputeAgent* ne doit en aucun cas bloquer l'évaluation

d'un automate suivant le modèle `Task`. Dans ce but, la variable globale `maxCATimeExec` (cf. Figure 3.22), qui correspond au temps maximum que peut mettre un *ComputeAgent* pour exécuter son code, est utilisée par le modèle `ComputeAgent` pour empêcher un automate de ce type de rester bloqué indéfiniment dans sa localité `Initialized`. L'horloge `execTime` permet de connaître le temps d'exécution du code de calcul. De son coté, la localité `CA_Initialized` du modèle `Task` définit l'invariant `h < maxCATimeExec` pour éviter un automate de ce type d'attendre indéfiniment une réponse du *ComputeAgent* associé. En effet, dans le cas où le *ComputeAgent* aurait été arrêté au cours de son évaluation, le modèle `Task` définit une transition pour revenir dans la localité `Pending` et ainsi redevenir disponible pour un autre agent $Worker$.



(a) Modèle d'automate `ComputeAgent`

(b) Modèle d'automate `User`

FIGURE 3.34 – Modèles d'automate `ComputeAgent` et `User` : résultats des π-transformations des terme *ComputeAgent* (eq. 3.9) et $User$ (eq. 2.62).

3.3.1.4 Transformation du terme principal

L'ajout d'un cas de calcul à l'agent *CaseDirectory* est réalisé par une émission d'ordre supérieur sur le canal *add*, celui-ci étant un paramètre formel du terme *CaseDirectory* (eq. 2.61). Le terme $User$ (eq. 2.62) modélise un utilisateur de l'espace de travail qui ajoute un cas de calcul. Le résultat de la π-transformation de ce terme est donné par la figure 3.34b. La cas de calcul ajouté est représenté par une instance du modèle d'automate `ComputationCase` (cf. Figure 3.26) qui a pour identifiant `idCC1` (cf. Figure 3.22).

Nous transformons le terme principal de notre spécification $MCA\pi Spec$ en appliquant la règle 3.1 sur le terme *System* (eq. 2.63). Le résultat de cette π-transformation est présenté par la figure 3.35 avec le modèle d'automate `System`. Celui-ci utilise les identifiants `idCD` et `idUser1` ainsi que le type `id_w` présents dans les déclarations globales du système UPPAAL (cf. Figure 3.22).

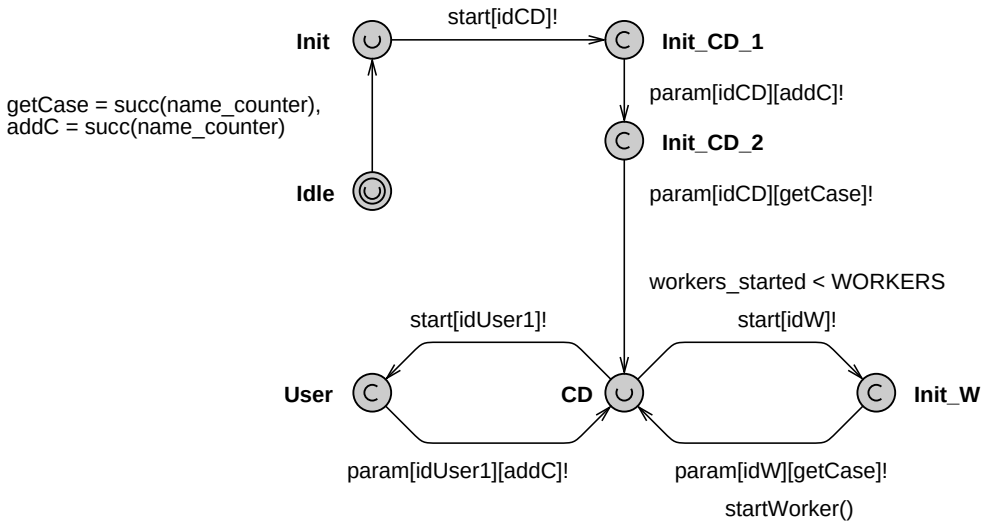


FIGURE 3.35 – Modèle d'automate `System` : résultat de la π-transformation du terme principal *System* (eq. 2.63)

3.3.2 Vérification du système

Nous avons mis en valeur des propriétés de transparence à partir des spécifications écrites au chapitre 2 : transparence de localisation et transparence d'échelle. Ces propriétés doivent être préservées dans l'implantation qui est faite au chapitre 4. Aussi, nous nous intéressons à leur preuve par model-checking.

3.3.2.1 Preuve de la transparence de localisation

Ramené à notre contexte de simulation, la transparence de localisation signifie que le terme $User$ ne connait pas la liste des $Workers$ qui vont intervenir pour traiter le cas de calcul qu'il vient d'ajouter. Ainsi, à un niveau d'abstraction supérieur, on souhaite établir que l'utilisateur de la plate-forme de simulation pense être le seul utilisateur, même si un autre utilisateur présent sur la plate-forme ajoute un cas de calcul et que ce dernier est évalué en parallèle. Pour établir cette propriété, nous établissons que :

- `A[] Master(7).Finished imply (forall (i : id_t)Task(i).Collected)` : pour toutes les exécutions, un cas de calcul considéré comme évalué implique que toutes les tâches qui le composent ont été collectées.
- `(Worker(3).On_Error and Worker(3).idCA == 13)--> (Worker(2).CA and Worker(2).idCA == 13)` : pour toutes les exécutions, lors de la défaillance d'un terme $Worker$ (modélisé par `Worker(3).On_Error`), la tâche en cours de traitement par celui-ci (modélisé par `Worker(3).idCA == 13`) est toujours remise à disposition afin d'être traitée par un autre $Worker$ (modélisé par `Worker(2).CA and Worker(2).idCA == 13`).

Au final, l'utilisateur ne sait pas comment la plate-forme de simulation traite les erreurs et assure ainsi une meilleure qualité de service.

3.3.2.2 Preuve de la transparence d'échelle

Notre plate-forme offre la possibilité du partage de ressource de calcul entre plusieurs cas de calcul, mais elle offre aussi, et surtout, la possibilité de s'adapter aux ressources disponibles. Bien sûr, le lecteur attentionné et averti pense avant tout aux pannes et autres défaillances du logiciel. Mais il existe aussi la possibilité inverse qui est d'augmenter les ressources de calcul. Cela se produit, par exemple, lorsqu'un cas de calcul se termine et libère ses ressources de calcul. Cette situation peut aussi se produire lorsque de nouveaux $Worker$ sont créés. Dans tous les cas, notre souci de transparence d'échelle nous conduit à nous interroger sur le comportement de l'évaluation d'un cas de calcul en cours de traitement.

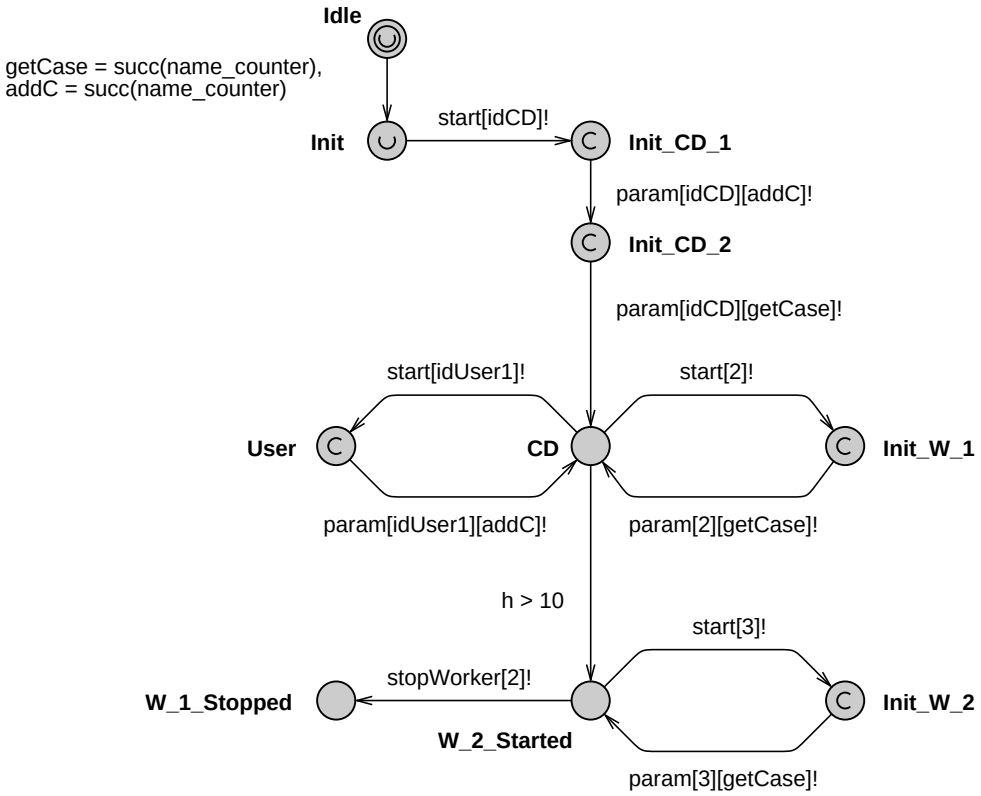


FIGURE 3.36 – Modèle d'automate `System_2` modélisant le démarrage et l'arrêt d'un agent $Worker$ après le début de l'évaluation d'un cas de calcul.

Pour écrire les propriétés permettant de vérifier cette transparence d'échelle, nous écrivons le modèle d'automate `System_2` (cf. Figure 3.36) afin de modéliser l'ajout d'un terme $Worker$ après le début d'un cas de calcul. La garde `h>10` permet d'être certain que le cas de calcul sera ajouté au $CaseDirectory$ avant que le terme $Worker$, modélisé par un automate ayant la valeur 3 comme identifiant, soit démarré. Enfin, la transition étiquetée par l'action de synchronisation `stopWorker[2]!` modélise l'arrêt d'un agent $Worker$ défini par un automate ayant la valeur 2 comme identifiant. Nous établissons alors les propriétés suivantes :

– `(Master(7).nb_th_to_stop == 1 and System.Init_W_2)--> Worker(3).CA` : il existe une exécution qui

lors de l'ajout d'un terme $Worker$ (modélisé par `System.Init_W_2`) au cours de l'évaluation d'un cas de calcul (modélisé par `Master(7).nb_th_to_stop == 1`) aboutit à l'occupation de ce $Worker$ par une tâche du cas de calcul (modélisé par `Worker(2).CA`).

– `(Master(7).nb_th_to_stop == 1 and System.W_1_Stopped)--> Master(7).Finished` : pour toutes les exécutions, la suppression d'un terme $Worker$ (modélisé par `System.W_1_Stopped`) pendant l'évaluation d'un cas de calcul (modélisé par `Master(7).nb_th_to_stop == 1`) est sans incidence sur l'évaluation complète du cas de calcul (modélisé par `Master(7).Finished`).

En conclusion, la variation du volume de ressource de calcul est invisible pour l'utilisateur modélisée dans notre spécification. Notre propriété est donc assurée.

3.4 Conclusion

La définition complète d'une transformation d'une spécification en π-calcul d'ordre supérieur vers un réseau d'automates temporisés constitue un résultat offrant une aide à la preuve de propriétés temporelles. C'est la possibilité d'appliquer à d'autres spécifications, écrites via une algèbre de processus, une étude de propriétés temporelles.

Sur le plan du cycle de vie d'un projet informatique, cette transformation $\mathcal{T}_S$ est une étape vers l'automatisation de notre démarche, afin de consacrer plus de temps sur l'écriture de propriétés et l'évaluation de leurs preuves.

Nous nous sommes intéressés à deux propriétés de transparence qui sont essentielles à notre projet. A l'heure du cloud computing, la transparence de localisation est non seulement une propriété à la mode mais surtout elle manifeste la volonté des utilisateurs à se détacher de contraintes matérielles datant du passé telles que le placement de code sur un processus ou le découpage des données en fonction d'une architecture. Nous avons établi cette propriété par utilisation de notre réseau d'automates temporisés et l'emploi d'un outil reconnu dans le monde de la preuve par model-checking. Nous validons ainsi la possibilité d'établir des propriétés liées à la mobilité de code via un réseau d'automates.

La transparence d'échelle revêt ainsi un caractère essentiel dans la tolérance aux pannes. En effet, nous souhaitons tous, en tant qu'utilisateur, ne pas voir perturber notre cas de calcul par la détérioration d'une ressource de calcul. Nous avons établi la preuve de ceci par l'étude générale de la transparence d'échelle. Nous constatons ainsi que notre système est apte à prendre en compte de nouvelles ressources de calcul et à les exploiter.

Dans le chapitre suivant, nous nous appuyons sur les deux chapitres que nous venons de présenter pour réaliser une implantation de l'architecture logicielle qui y est définie.

Chapitre 4

Implantation d'un framework tolérant aux pannes pour la résolution de cas de calcul numérique

Sommaire

Afin de valider les concepts présentés dans les chapitres 2 et 3, nous avons réalisé le framework *MCA* (pour *M*obile *C*omputing *A*rchitecture), développé en Java, permettant la résolution de cas de calcul numériques dans un environnement distribué hétérogène.

Avant de présenter notre framework dans la suite de ce chapitre, il est important de souligner que les systèmes distribués sont fondamentalement différents des systèmes non-distribués. En effet,

dans un système distribué, il existe des situations dans lesquelles des membres du système ne sont plus en mesure de communiquer avec les autres membres du même système. Cela arrive en général pour deux raisons principales :
– Soit parce que l'un des membres de la communauté est en panne ;
– Soit parce que la connexion entre les membres de la communauté ne fonctionne plus.
Ce type de défaillance, dite partielle -c'est à dire la défaillance d'une partie du système-, peut survenir à tout moment et peut être intermittente ou de longue durée.

Un des principaux challenges de la mise en place d'un système distribué est d'être tolérant à ces différentes pannes. En effet, la propriété de réagir face à une défaillance dans un tel système complique considérablement le travail des développeurs. En contrepartie, rendre un système distribué tolérant aux pannes est une marque de qualité et un atout indéniable pour se différencier des autres systèmes du même type. Les composants mis en relation dans ce type de système proposent des ressources ou des services aux autres composants du système. La panne d'un ou plusieurs de ces composants peut conduire, par exemple, à des ressources inutilisées non libérées ou à des services qui continuent de s'exécuter alors que le client du service n'attend plus de réponse. Ces situations entrainent un système distribué vers une consommation inutile des ressources ou, plus grave encore, à un blocage total du système. Du point de vue de l'utilisateur final, cela signifie ne par voir ces pannes et donc ne pas perdre le temps de calcul déjà consommé (transparence de panne).

Dans ce chapitre, nous commençons par la présentation générale de l'architecture logicielle définie par notre framework *MCA* (Section 4.1). Ensuite, nous listons les différents outils et paradigmes proposés par l'API Jini™ et utilisés par notre framework MCA, comme la notion de services et de mobilité (Section 4.2) et la technologie *JavaSpaces* (Section 4.3). La section 4.4 détaille les différents éléments qui composent cette architecture. Enfin, la section 4.5 propose un bilan des solutions offertes par le framework *MCA*.

4.1 Présentation générale de l'architecture *MCA*

4.1.1 Apport de notre modèle formel

Les résultats obtenus dans le chapitre 2 ont abouti à la définition des exigences pour une plate-forme logicielle de calcul numérique où les transparences de localisation, d'échelle et de concurrence sont respectées. Ainsi, les différents termes π-calcul écrits dans le chapitre 2 se concrétisent par les composants présentées dans ce chapitre (cf. Figure 4.1) :
– Le terme $CaseDirectory$ (eq. 2.61) se concrétise par le service *MCAService* ;
– Le terme $Worker$ (eq. 2.64) se concrétise par un agent *MCAWorker* et le terme $Worker^m$ (eq. 2.37) se concrétise par l'ensemble des agents $MCAWorker$;
– le terme $ComputeAgent$ (eq. 2.41) se concrétise par un agent mobile de type *ComputeAgent* ;

– Le terme *ComputationCase* (eq. 2.58) se concrétise par le composant *ComputationSpace*. Celui-ci est un *space* et concrétise également les annuaires modélisés par les termes $TaskDirectory$ (eq. 2.20), $DHDirectory$ (eq. 2.40) et $PropertyDirectory$ (eq. 2.49). Le terme $Master$ (eq. 2.60) est lui concrétisé par un agent mobile d'un type de *ComputeAgent*. Enfin, le rôle modélisé par le terme $CaseHanler$ (eq. 2.59) se concrétise par les fonctionnalités offertes par l'utilisation d'un *space*.

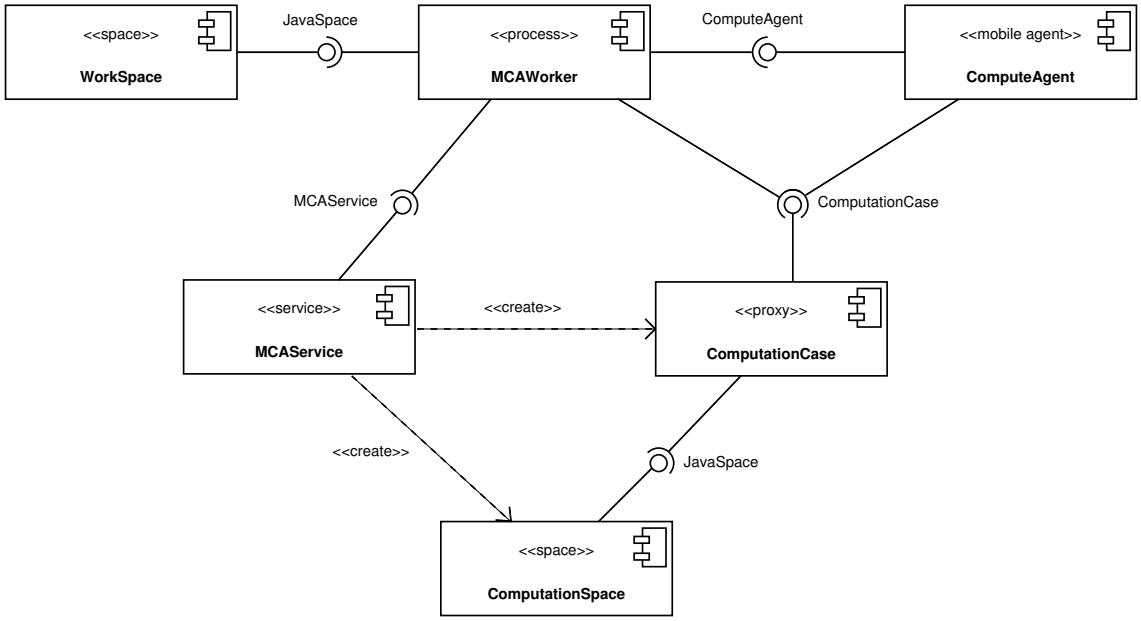


FIGURE 4.1 – Diagramme de composants de l'architecture logicielle définie par le framework *MCA*.

Notons que le composant $WorkSpace$ n'a pas été formellement modélisé mais celui-ci est utilisé dans notre architecture pour répondre à une exigence de tolérance aux pannes des agents *MCAWorker*.

4.1.2 Survol de notre architecture logicielle

Notre framework *MCA* permet de mettre en place une architecture logicielle dédiée à la résolution de cas de calcul numérique. Chaque cas de calcul est associé à un *ComputationSpace* qui contient les différents éléments nécessaires à la résolution du cas. L'ensemble des cas de calcul, chacun représenté par un *ComputationSpace*, forme la plate-forme de calcul MCA. Cette plate-forme est accessible par le service *MCAService* dont plusieurs instances sont actives afin de rendre ce service tolérant aux pannes. Ce service offre à un utilisateur la possibilité d'ajouter, de récupérer ou de supprimer un cas de calcul de la plate-forme. De plus, les agents *MCAWorker* se connectent à la plate-forme de calcul par le service *MCAService*. Ces agents ont alors la possibilité de participer à la résolution d'un cas de calcul via le composant *ComputationCase* qui joue le rôle de *proxy*. Si le nombre d'agents *MCAWorker* connectés à la plate-forme de calcul est supérieur à

la demande de ressources de calcul, alors les agents *MCAWorker* inactifs sont considérés comme disponibles (transparence de localisation). Ces derniers seront utilisés si un cas de calcul venait à demander de nouvelles ressources ou si un nouveau cas de calcul était ajouté à la plate-forme de calcul. La figure 4.2) propose une vue d'ensemble de cette architecture avec les composants *ComputationSpace*, *MCAService* et *MCAWorker*. Une vue plus précise est fournie par la figure 4.16 montrant les communications entre un agent *MCAWorker* et le service *MCAService*.

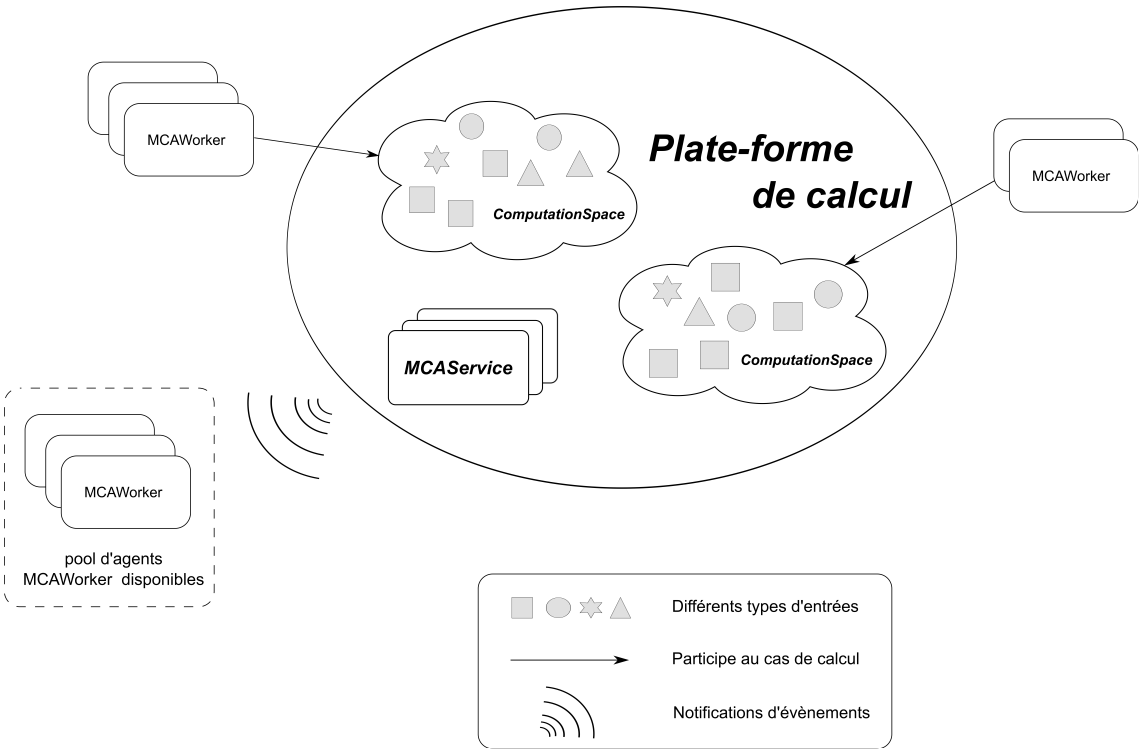


FIGURE 4.2 – Vue d'ensemble de l'architecture proposée par le framework *MCA*.

Les composants *ComputationSpace* et *WorkSpace* (mémoire locale d'un agent *MCAWorker*) sont des mémoires partagées, appelées « *spaces* » (cf. Section 4.3). Ils constituent le socle nécessaire pour rendre l'architecture définie par notre framework MCA tolérante aux pannes. La figure 4.1 présente les différents composants qui communiquent au sein de cette architecture.

Enfin, la résolution d'un cas de calcul est réalisée par les agents *MCAWorker*. Ceux-ci traitent les différentes tâches présentes dans le *ComputationSpace* associé au cas de calcul. À chaque tâche est associé un agent mobile de type *ComputeAgent* (cf. Section 4.4.3). Ce dernier contient le code à exécuter pour le traitement d'une tâche. Son aspect mobile lui permet de migrer vers la machine où est exécuté l'agent *MCAWorker* traitant la tâche. Le code est alors exécuté en local sur la machine de l'agent *MCAWorker*.

La suite du chapitre présente les différents éléments de notre framework en commençant par le socle logiciel sur lequel il se fonde.

4.2 Un environnement d'exécution adaptatif

Dans cette section nous présentons le socle logiciel sur lequel notre framework *MCA* se fonde. Dans un premier temps, nous abordons la technologie Jini™ (Section 4.2.1) car l'architecture *MCA* s'appuie essentiellement sur celle-ci avec l'utilisation des différents services de base qu'elle propose. Le service *MCAService* est un service *Jini* (Section 4.2.1.1) et, à ce titre, il dispose des mêmes caractéristiques que tous les autres services *Jini*, en particulier la possibilité de définir une politique de sécurité lors des invocations distantes de ses opérations (Section 4.2.1.4). Dans un second temps, nous présentons la mise en place d'un système d'agents mobiles utilisé pour les composants *ComputeAgent* et *ComputationCase* de l'architecture *MCA* (Section 4.2.2).

4.2.1 L'architecture orientée services offerte par la technologie Jini™

La technologie Jini™ [WT00] a été définie pour aider les développeurs de systèmes distribués à traiter de manière simple une des principales caractéristiques de ce type de système : l'adaptation au changement. Le changement dans une application distribuée se traduit par l'arrivée ou le départ de composants qui constituent le système. Un des problèmes qui en découle, et qui doit être absolument pris en compte, est la défaillance partielle, c'est à dire la défaillance d'une partie, et non de tout le système. En effet, dans un système distribué, des composants peuvent s'exécuter alors que d'autres sont arrêtés, ou alors tous les composants peuvent être opérationnels mais la connexion réseau peut être défaillante.

À ce jour, plusieurs implémentations de la spécification Jini™ sont disponibles : *JSC*, *Seven*, *Servicehost*, *Rio*, *Harvester*, *H2O* et *CoBRA*. L'ensemble de ces implémentations ont été présentées par Svetozar Misljencevic dans [Mis06]. Pour notre architecture logicielle, notre choix s'est porté vers l'implantation de référence car elle a déjà été utilisée par notre équipe sur d'autres travaux [Ber09] et l'expérience acquise ne peut être qu'un avantage. Cette implantation est développée par *Sun*, se nomme *Jini* [New06] et est disponible via le *Jini starter kit*.

4.2.1.1 Définition d'un service *Jini*

Jini™ est basé sur la notion de service et permet de créer des systèmes présentant une architecture orientée service. Un système adoptant l'architecture proposée par la technologie Jini™ est donc composé de divers services disponibles sur le réseau. Chaque service est accessible via un objet mandataire, appelé « *proxy* ». Lorsqu'un client désire utiliser un service, il doit en premier lieu obtenir un *proxy* de ce service. Le client invoque ensuite les méthodes à travers ce *proxy*. Ce dernier exécute alors la méthode désirée sur le service distant en prenant en charge les communications à travers le réseau.

L'arrivée d'un nouveau service, ou la migration d'un service vers une autre localité, sont des événements récurrents dans un système distribué. Malgré cela, un client doit pouvoir trouver

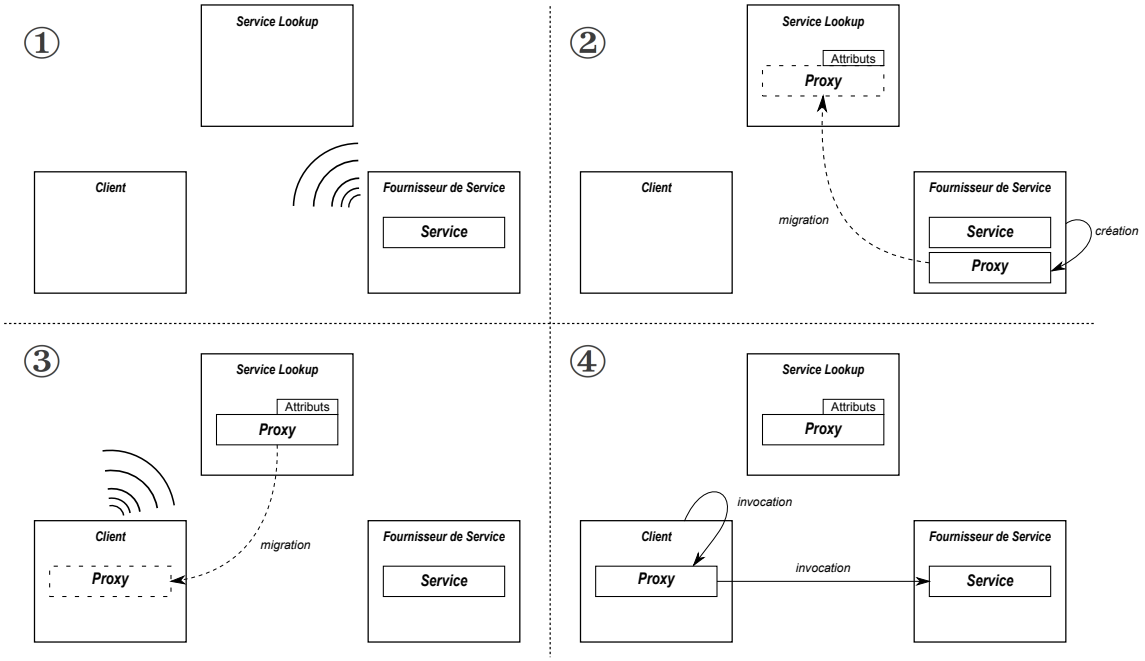


FIGURE 4.3 – Les protocoles mis en œuvre par la technologie Jini™ . ① *Découverte* : le fournisseur du service recherche le service *Lookup*. ② *Adhésion* : Le fournisseur crée et enregistre le proxy du service dans l'annuaire. ③ *Recherche* : Le client recherche le service en fonction de ses attributs. Le service *Lookup* lui transmet une copie du proxy associé. ④ *Invocation* : Le client invoque le service via le proxy associé.

facilement un service sur le réseau. Jini™ a une solution : le service *Lookup*. Celui-ci propose un annuaire dans lequel chaque service disponible sur le réseau enregistre son *proxy*. Le service *Lookup* est utilisé par un client pour découvrir des services sans connaître au préalable leur localisation. Si un client souhaite utiliser un service pour la première fois, il recherche le *proxy* correspondant à ce service dans un annuaire (*Lookup*). Soit le client connaît la localité d'un annuaire soit il utilise un ensemble de protocoles de découverte (*discovery protocols*) fournis par Jini™ pour trouver l'annuaire sur le réseau. Une fois que le client obtient le *proxy* du service désiré, il utilise ce *proxy* pour communiquer directement avec le service, et cette fois-ci sans la participation de l'annuaire. La figure 4.3 propose les différents protocoles que nous venons de décrire.

4.2.1.2 La notion de bail

Pour répondre aux possibles défaillances partielles d'un système distribué, la technologie Jini™ apporte une solution simple avec la notion de **bail**. Le concept de base est défini comme suit : il est préférable de définir un intervalle de temps restreint durant lequel un service peut être accessible plutôt que de laisser indéfiniment ce service accessible. Quand un client désire utiliser un service *Jini*, il doit demander au service de se maintenir dans un état durant lequel il pourra y accéder, le service fournit alors au client un bail. Ce bail est la période de temps durant laquelle le

service accepte de garder, dans la limite de ses possibilités, un état accessible pour le client. Cette période est déterminée par le service ou négociée entre le client et le service. Durant cette période, le client peut annuler le bail, ce qui permet au service de libérer les ressources qu'il utilisait pour maintenir un état correct vis-à-vis du client. Mais le client peut aussi vouloir renouveler le bail, dans ce cas le service peut renouveler le bail avec la période de temps demandée (ou avec une période plus courte[9]) ou refuser la demande de renouvellement. Si la période de temps du bail n'est pas renouvelée, alors le service est libre de libérer les ressources associées au maintien de l'état.

L'utilisation de baux évite à un service *Jini* d'utiliser des ressources inutilement. Par exemple, si un client demande à un service de se maintenir dans un état accessible et que ce même client a une panne, alors ce client sera incapable de renouveler le bail. Le bail expirera et le service pourra alors libérer les ressources utilisées. De la même manière, si le service et le client fonctionnent normalement mais que le réseau entre eux connait une défaillance, alors le client sera incapable de renouveler le bail ce qui permettra à l'expiration du bail de libérer les ressources utilisées inutilement.

La notion de bail est présente lors de l'enregistrement d'un service, comme par exemple le service *MCAService*, dans l'annuaire que propose le service *Lookup*. Lors de son enregistrement, le fournisseur du service obtient un bail créé par l'annuaire et lié au service qu'il vient d'enregistrer. Le service reste enregistré dans l'annuaire tant que le fournisseur de ce service renouvelle ce bail. Si le fournisseur du service vient à avoir une défaillance, le bail arrivera alors à expiration et le proxy du service sera supprimé de l'annuaire.

4.2.1.3 La définition d'un modèle d'invocation distante

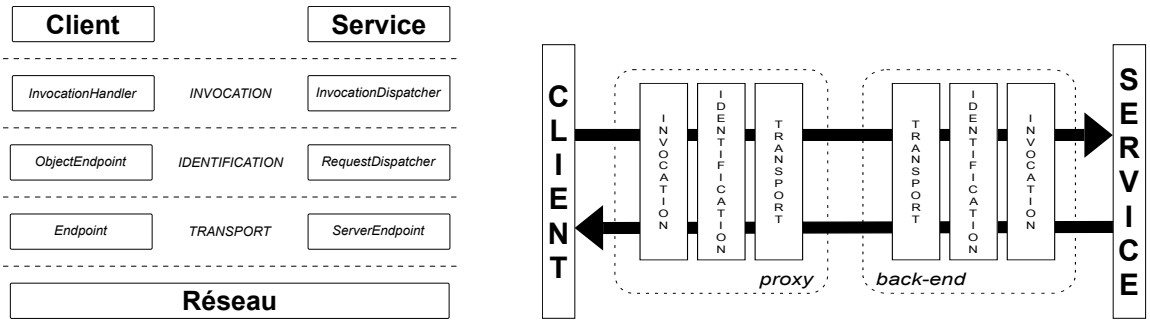
Le langage Java propose l'API *Remote Method Invocation* (*RMI*) pour effectuer des invocations de méthodes sur des objets distants. Cette API définit un modèle de programmation qui facilite la communication d'objets Java s'exécutant sur deux machines virtuelles distinctes. Cette communication est réalisée à l'aide de deux protocoles : le protocole *JRMP* (pour *Java Remote Method Protocol*) ou le protocole *IIOP* (pour *Internet Inter-Orb Protocol*).

Jini™ étend ce modèle avec le modèle *Jini ERI* (pour *J*ini *E*xtensible *R*emote *I*nvocation). Comme le montre la figure 4.4a, le modèle défini par *Jini ERI* se compose de trois couches : une couche d'invocation, une couche d'identification et une couche de transport.

– La *couche transport* fait communiquer les requêtes et les réponses à travers le réseau. Cette couche permet l'envoi d'une requête cliente au service à l'aide d'une instance de la classe `Endpoint` et contrôle ainsi son traitement par le service via une instance de la classe `ServerEndpoint` .

9. Nous ne rentrons volontairement pas dans le détail de la configuration Jini™ , mais lors de la configuration d'un service Jini™ il est possible de définir une période temps maximum pour un bail entre le service et un client.

– La *couche identification* permet de distinguer des objets distants lors de l'invocation de méthodes. Ainsi, coté client, une instance de `ObjectEndpoint` contient l'identifiant de l'objet distant ainsi qu'un point d'entrée pour communiquer les requêtes vers cet objet distant. Coté service, une instance de `RequestDispatcher` contient une table de correspondance associant un identifiant client au *proxy* utilisé par ce client.
– La *couche d'invocation* dirige le processus d'invocation distante. Coté client, une instance de `InvocationHandler` sérialise l'appel de la méthode avec ses paramètres éventuels et désérialise la réponse retournée par le service. Coté service, une instance de `InvocationDispatcher` désérialise les invocations faîtes par le *proxy* associé et sérialise la réponse qui est envoyée au client.



(a) (b)

FIGURE 4.4 – Les différentes couches du modèle *Jini ERI*.

Dans une application Jini™ , le client invoque une méthode du service désiré via un *proxy*. C'est le fournisseur du service qui lie le service avec un modèle d'invocation distante (*JRMP*, *IIOP* ou *JERI*) à l'aide d'un « service exporter ». Celui-ci crée, pour le modèle d'invocation désiré, deux parties distinctes qui permettent de faire communiquer un client avec le service :
– Une partie cliente, le *proxy* de la Figure 4.4b, qui est enregistrée sur un annuaire et utilisée par les clients du service ;
– Une partie serveur, le *back-end* de la Figure 4.4b, qui reste lié au service.

L'interface `net.jini.export.Exporter` généralise les différents moyens d'« exporter » un service. La création d'un *proxy* est définie par une implantation de cette interface. L'API proposée par Jini™ définit plusieurs implantations avec les classes `net.jini.jrmp.JrmpExporter` et `net.jini.iiop.IiopExporter` pour un modèle d'invocation distante utilisant respectivement les protocoles *JRMP* et *IIOP* définis par le modèle RMI.

Dans l'architecture proposée par le framework *MCA*, nous utilisons le modèle *Jini ERI* pour les invocations distantes des méthodes proposées par les services *MCAService*, *Lookup* et *JavaSpaces*. La classe `BasicJeriExporter` facilite la mise en place du modèle *Jini ERI* pour la couche d'invocation distante. Une instance de la classe `BasicJeriExporter` est créée pour chaque service proposé par l'architecture *MCA* et chaque « exporter » contient les informations permettant de contrôler l'implantation de chaque couche du modèle *Jini ERI* que ce soit du coté client ou du coté serveur.

4.2.1.4 Mise en place d'une politique de sécurité

Le modèle d'invocation de méthodes sur des objets distants demande de définir une politique de sécurité pour assurer les propriétés suivantes au sein de notre architecture :

- *Authentification* - Elle assure que chaque composant de l'architecture soit authentifié, qu'il soit un client ou un service. De cette façon, chaque composant est capable d'identifier le composant avec lequel il communique. Par exemple, un agent *MCAWorker* doit être authentifié avant de participer à la résolution d'un cas de calcul et un agent *MCAWorker* doit être sûr qu'il demande bien à un service *MCAService* authentifié.
- *Autorisation* - Un fois authentifié, un composant peut se voir accorder des droits. Par exemple, le service *MCAService* d'une plate-forme de calcul autorise seulement les utilisateurs authentifiés en tant qu'administrateur à ajouter un cas de calcul à la plate-forme.
- *Intégrité* - Elle assure que les données soient complètes et exactes après le transfert par le canal de communication. Comme par exemple, le résultat d'une tâche ne doit pas être corrompu.
- *Confidentialité* - Elle assure que les données ne soient lisibles uniquement par les composants autorisés. Par exemple, seuls les agents *Worker* participant à un cas de calcul pourront lire les données de ce cas.

Pour définir notre politique de sécurité lors de l'utilisation des différents services (*MCAService*, *ComputationSpace*, *WorkSpace*) au sein de notre architecture MCA, nous nous appuyons sur le protocole *Jini ERI* que nous venons de présenter. Jini™ part de l'hypothèse qu'un réseau n'est pas nécessairement sécurisé. Pour remédier à cela, Jini™ étend le modèle de sécurité de la plate-forme Java pour définir un nouveau modèle. Les services et les clients (de ces services) peuvent l'utiliser pour agir de façon sécurisée dans un réseau qui ne l'est pas.

Nous utilisons l'*API JAAS* (pour *Java Authentification and Authorization Service*) [JP01] pour autoriser un composant de s'exécuter sous une certaine identité (dite « *Principal* »). Chaque composant de notre architecture est donc exécuté dans un contexte authentifié. L'identité est définie par un certificat numérique *X.509*. Chaque certificat est signé par une autorité de confiance (CA), ce qui permet d'assurer la validité de l'identité du composant avec lequel on communique.

Sécurité coté serveur

Lors de la création d'un « service exporter » (voir le code de la figure 4.5), il est possible de définir la politique de sécurité utilisée pour l'invocation des méthodes de ce service. Deux éléments sont fournis pour paramétrer ce « service exporter » :

- Une fabrique de type `InvocationLayerFactory` (`BasicILFactory` dans la figure 4.5) qui définit comment créer un *proxy* (contenant un gestionnaire d'invocation de type `InvocationHandler`) destinés aux clients et un répartiteur d'invocation de type `InvocationDispatcher` pour le service.
- Un point d'entrée coté service (de type `ServerEndpoint`) pour la couche transport qui est aussi utilisé pour créer les points d'entrée (de type `Endpoint`) pour les clients de ce service.

Nous utilisons le protocole *TLS/SSL* (voir encadré sur le protocole *TLS/SSL*) dans la couche transport du modèle *Jini ERI*. La classe `SslServerEndpoint` permet d'utiliser *TLS/SSL* entre un client et le service désiré. Une instance de cette classe est passée au constructeur de la classe `BasicJeriExporter` afin que le *proxy* de chaque service de l'architecture *MCA* puisse utiliser *TLS/SSL*. Avec l'utilisation de ce protocole, les contraintes listées dans le tableau 4.1 sont implicitement respectées. Les communications entre les différents composants de l'architecture *MCA* s'appuient de ce fait sur les propriétés demandées : authentification, autorisation, intégrité et confidentialité.

```
1 SslServerEndpoint serviceEndpoint = SslServerEndpoint.getInstance(HOST,0);
2 MethodConstraints serviceConstraints = ...; // voir table 4.1
3 Class<?> permissionClass = MCASpacePermission.class;
4 BasicILFactory  serviceILFactory = new BasicILFactory(serviceConstraints, permissionClass);
5 Exporter exporter = new BasicJeriExporter(serviceEndpoint, serviceILFactory)
```

FIGURE 4.5 – Code source pour créer un « service exporter ».

Le protocole TLS/SSL

SSL (*Secure Socket Layer*) est un protocole standard fonctionnant sur TCP/IP. SSL suit le modèle client-serveur et fournit les services sécurisés suivants [FKK11] :
– Authentification du serveur et du client avec l'utilisation de certificat numérique *X.509*
– Confidentialité des données échangées à l'aide de chiffrement symétrique (comme par exemple l'algorithme *AES*[a]) et asymétrique (comme par exemple l'algorithme *RSA*[b])
– Intégrité des données échangées à l'aide d'une fonction de hachage comme le *SHA-1*[c]
Le protocole SSL peut aussi fonctionner sous d'autres protocoles de TCP/IP comme par exemple HTTP. SSL v3 est le standard SSL le plus généralement utilisé même si le protocole TLS (*Transport Layer Security*) élaboré par l'IETF (*Internet Engineering Task Force*) devient le nouveau nom du protocole SSL.

a. *AES* pour *Advanced Encryption Standard*
b. *RSA* pour *Rivest Shamir Adleman*
c. *SHA* pour *Secure Hash Algorithm*

Sécurité coté client

Avant de pouvoir invoquer des méthodes à travers un *proxy* reçu depuis le service *Lookup*, le client doit « préparer » ce *proxy* afin de s'assurer que les invocations se feront dans un contexte sécurisé. La « préparation » d'un *proxy* est effectuée par la méthode `prepareProxy` déclarée dans l'interface `ProxyPreparer` (cf. Figure 4.6) et suit les étapes suivantes :

– *Vérifier le proxy.* Le client commence par vérifier si le *proxy* est digne de confiance car un *proxy* peut avoir été reçu à partir d'une source non fiable ce qui peut représenter un danger. En effet si le client ne fait pas cette vérification alors il y a le danger que le proxy puisse ignorer les

contraintes imposées par le client et effectuer un transfert de données en clair (non chiffré) ou alors déformer l'identité du client ou du serveur par exemple. L'utilisation du protocole *TLS* dans l'architecture *MCA* assure la fiabilité des *proxies* utilisés.

– *Accorder des permissions.* Une fois que le client a vérifié le *proxy*, il peut lui accorder des permissions supplémentaires (en plus de celles déjà accordées au code non fiable), pour permettre aux futures invocations à travers ce proxy de fonctionner parfaitement. Par exemple, un *proxy* peut exiger une permission de type `AuthenticationPermission` afin qu'il puisse s'authentifier auprès du serveur. Il paraît donc évident qu'un client doit accorder des permissions supplémentaires à un *proxy* uniquement s'il lui fait entièrement confiance. Un *proxy* non fiable pourrait abuser de ses nouveaux droits pour causer de graves dommages sur la machine du client.

– *Définir des contraintes.* Le client peut appliquer des contraintes au *proxy*. Une contrainte est une exigence spécifique sur le comportement désiré lors de l'invocation d'une méthode sur le service distant à travers ce *proxy*. Il est important de noter que chaque contrainte exprime ce que représente la contrainte sur le contexte d'exécution mais non comment elle est vérifiée. C'est en effet le *proxy* qui est responsable d'utiliser les protocoles de transport appropriés pour satisfaire les exigences du client. Le tableau 4.1 liste les différentes classes de l'API qui permettent de définir des contraintes sur les invocations de méthode à travers un proxy.

```
1 KeyStore keyStore = KeyStores.getKeyStore("file:keystore.worker", null);
2 X500Principal clientUser = KeyStores.getX500Principal("server", keyStore);
3 ServerMinPrincipal smp = new ServerMinPrincipal(clientUser);
4 InvocationConstraints ics =
5   new InvocationConstraints( new InvocationConstraint[]{Integrity.YES,ServerAuthentication.YES,smp}, null);
6 ProxyPreparer preparer =
7   new BasicProxyPreparer(true, new BasicMethodConstraints(ics), new Permission[]{});
```

FIGURE 4.6 – Code source pour préparer un *proxy* avant l'invocation des méthodes du service associé.

4.2.2 Implantation d'un système d'agents mobiles

Le concept de base d'une architecture définie par Jini™ est la notion de service. Un service enregistre son *proxy* dans un annuaire, proposé par le service *Lookup*, et un client vient chercher ce *proxy* pour communiquer avec le service correspondant. Jini™ propose une autre façon d'enregistrer un service dans l'annuaire. Il est en effet possible d'enregistrer directement un service sans passer par la création d'un *proxy*. Dans ce cas, l'objet déposé dans l'annuaire est l'instance d'une classe implantant directement ou indirectement l'interface `java.io.Serializable`. Le client qui récupère cet objet, via la méthode `lookup`, exécute alors localement le code du service. Comme le code s'est déplacé, on parle de *code mobile*. L'instance qui contient ce code mobile est appelée

Classe	Valeurs	Description
`Integrity`	`YES/NO`	Exige l'intégrité du contenu du message. Ainsi lors de l'appel d'une méthode distante, le proxy va s'assurer que les données échangées demeurent intactes et que le code à télécharger est accessible par des URLs (*codebase*) assurant elles-mêmes une intégrité du contenu qu'elles proposent.
`Confidentiality`	`YES/NO`	Exige la confidentialité du contenu du message. Le proxy va ainsi s'assurer que le message est chiffré et qu'il ne peut être compris que par les deux parties de la communication.
`ClientAuthentication`	`YES/NO`	Exige du client d'être authentifié et empêche ainsi l'invocation d'une méthode par un client « anonyme ».
`ClientMinPrincipal`	`Principal[]`	Exige du client de s'authentifier comme l'une des identités (`Principal`) définies.
`ServerAuthentication`	`YES/NO`	Exige du serveur d'être authentifié et empêche ainsi l'invocation d'une méthode sur un serveur « anonyme ».
`ServerMinPrincipal`	`Principal[]`	Exige du serveur de s'authentifier comme l'une des identités (`Principal`) définies.

TABLE 4.1 – Classes appartenant au package `net.jini.core.constraint` utilisées pour définir des contraintes d'invocation sur un *proxy*.

« *agent mobile* », ce qui est le cas du *ComputeAgent* de l'architecture *MCA* (cf. Figure 4.1).

4.2.2.1 Migration réactive versus migration proactive

La section 1.4 nous a présenté que deux types de migration sont possibles pour un agent mobile : une migration *réactive*, c'est à dire que l'agent se déplace suite à la demande d'un autre agent, ou une migration *proactive*, dans ce cas l'agent mobile décide lui-même quand et où il doit se déplacer.

Une migration dite *réactive* est semblable à l'utilisation d'un service comme nous l'avons décrit dans la section 4.2.1. L'agent mobile est enregistré dans un annuaire via le service *Lookup* avec des attributs pour faciliter sa recherche. Le client voulant exécuter cet agent mobile récupère alors l'agent en local et non un *proxy* comme c'est le cas pour un service *Jini* classique. La figure 4.7 décrit les différentes étapes de la migration d'un agent mobile utilisant une migration réactive. Cette pratique est illustrée par l'utilisation d'un agent de type *ComputeAgent* faisant partie de notre architecture (cf. Section 4.4.3).

Jini™ ne permet pas de créer un agent mobile dit « *proactif* », mais la possibilité de surveiller l'activité d'un service *Lookup* permet de simuler ce type d'agent mobile. Deux composants sont ici mis en jeu : le premier est l'agent mobile lui même qui a une (ou parfois plusieurs) tâche à réaliser et le second est l'hôte qui accueille cet agent mobile dit « *proactif* ». Un processus hôte offre un environnement d'exécution à un agent mobile et ce dernier ne peut se déplacer uniquement vers une localité qui dispose de ce processus hôte. Ce dernier est composé d'un service *Lookup* et d'un *Agent Listener* (instance de la classe `MobileAgentListener`) qui surveille l'activité du service *Lookup*, en particulier l'arrivée d'un agent mobile (instance d'une classe héritant de la classe

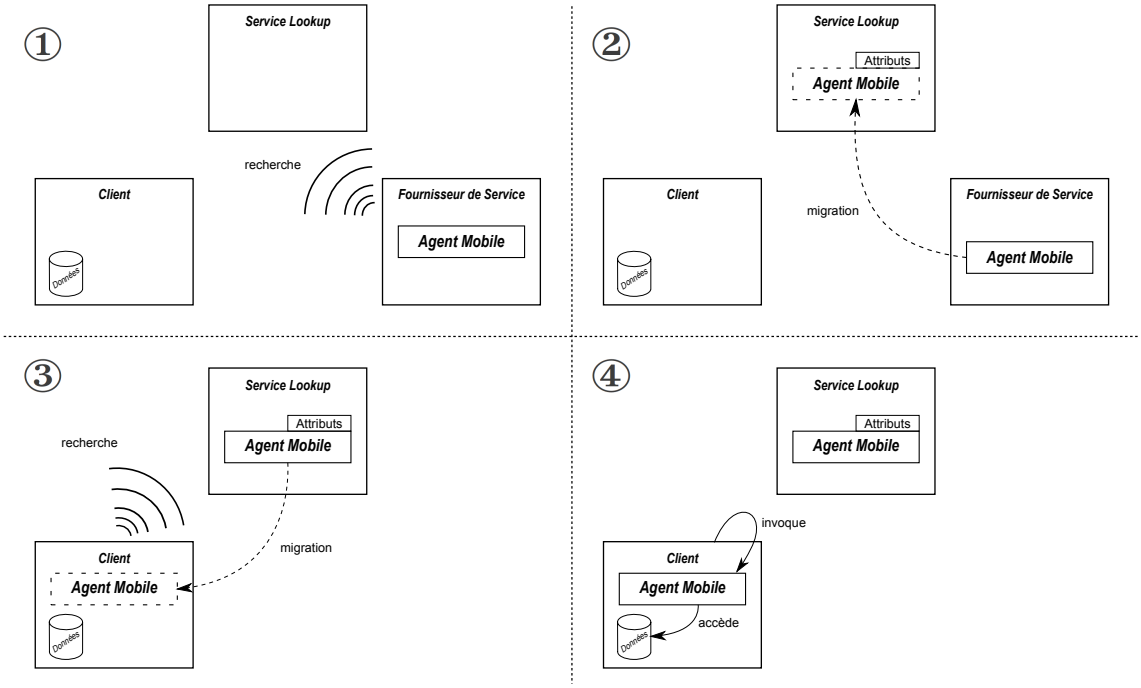


FIGURE 4.7 – Les différentes étapes d'une migration *réactive*. ① Le fournisseur du service recherche le service *Lookup*. ② Le fournisseur du service enregistre l'agent mobile dans l'annuaire en indiquant certains attributs pour faciliter sa recherche. ③ Le client recherche l'agent mobile en fonction de ses attributs. Le service *Lookup* lui transmet une copie de l'agent mobile associé. ④ Le client invoque une méthode de l'agent mobile. Le code est alors exécuté sur la machine du client.

abstraite `MobileAgent`). A l'arrivée d'un agent mobile, le processus *AgentListener* est prévenu (via la méthode `notify`) et peut exécuter le code de l'agent mobile via la méthode `execute`. A la fin de son exécution, l'agent mobile migre vers un autre hôte via sa méthode `move` (cf. Figure 4.8).

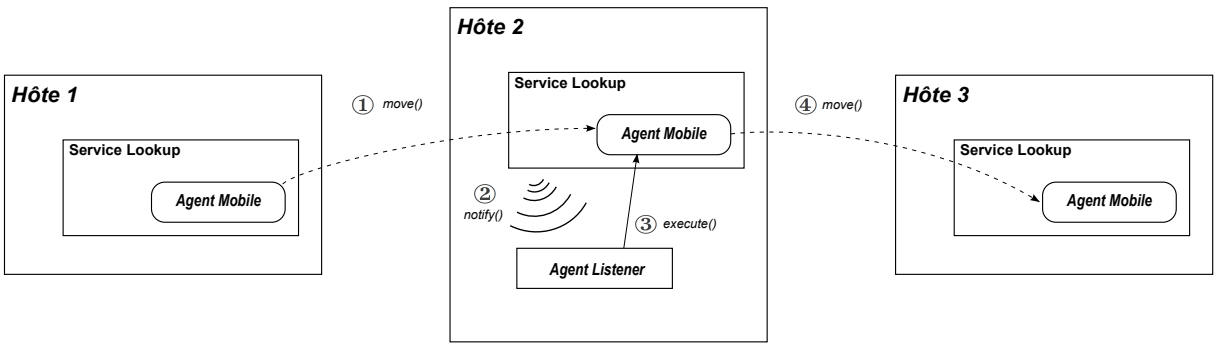


FIGURE 4.8 – Composants nécessaires à la mise en place d'un agent mobile *proactif*. ① L'agent mobile migre de l'*Hôte 1* vers l'*Hôte 2*. ② L'*AgentListener* est notifié de l'arrivée de l'agent mobile. ③ L'*AgentListener* de l'*Hôte 2* exécute le code de l'agent mobile. ④ A la fin de son exécution, l'agent mobile migre de l'*Hôte 2* ver l'*Hôte 3*.

4.2.2.2 Une politique de sécurité adaptée à la mobilité de code

L'accueil d'un agent mobile par un processus hôte est une action qui comporte un risque maximal (cf. Section 1.4.3). Parmi les différentes techniques proposées, la mise en place d'agents mobiles dans un environnement d'exécution comme celui proposé par *Java* permet de mettre en pratique les techniques suivantes :

La technique du bac à sable - L'utilisation du langage *Java* pour développer notre framework nous permet de profiter de la possibilité de pouvoir limiter les droits à un programme qui s'exécute dans une JVM. Ainsi, lors de l'exécution d'un agent mobile, l'hôte peut facilement limiter les possibilités de celui-ci (cf. Figure 4.9).

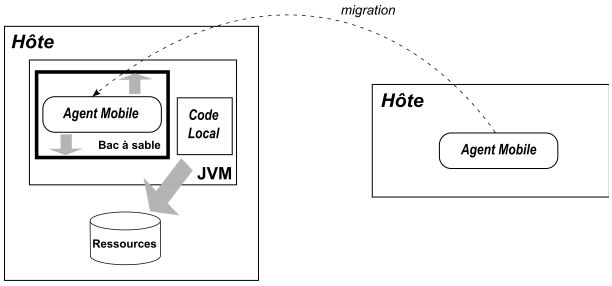


FIGURE 4.9 – La technique du bac à sable.

La signature du code - La signature du code intervient lors de la création d'un agent. Son créateur le signe numériquement. En réalité, il signe le fichier *jar* contenant la définition des classes Java. De cette façon, l'agent mobile est authentifié durant ses déplacements (cf. Figure 4.10). Dans notre cas, avec la technologie Jini™ , l'hôte télécharge la définition des classes de l'agent mobile à l'aide du *codebase* défini lors du déploiement de l'agent mobile via le service *Lookup*. Cette technique permet d'obtenir une authentification de haut niveau pour les hôtes. Elle assure aussi l'intégrité du code pour l'hôte visité. Une signature digitale sert donc de moyen de confirmation de l'authenticité de l'agent mobile, de son origine et de son intégrité.

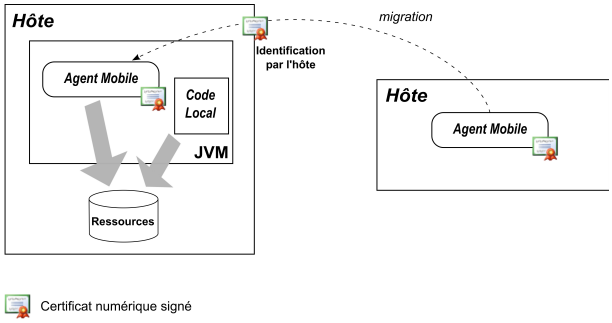


FIGURE 4.10 – Signature du code mobile.

Le contrôle d'accès - Un programme qui s'exécute sur un système doit accéder à des ressources pour réaliser sa tâche. Un agent mobile doit avoir un environnement d'exécution restreint pour des

raisons de sécurité. C'est pourquoi les accès aux ressources, locales à l'hôte, dont il a besoin pour traiter sa tâche, doivent donc être rigoureusement contrôlés. Pour améliorer les deux techniques précédentes, une politique de contrôle d'accès plus complexe est mise en place. La gestion des droits d'accès aux ressources pour un agent mobile passe par l'élaboration d'une politique de sécurité. Il est possible de choisir entre les politiques suivantes :

– L'hôte autorise l'accès à toutes les ressources pour tous les agents mobiles.
– Tous les agents mobiles sont soumis à la même politique sur l'hôte.
– Une négociation a lieu pour chaque agent mobile.

Il est essentiel pour un système hôte d'être capable d'authentifier les agents mobiles qu'il accueille. L'identité du signataire du code mobile permet de raffiner la définition de la politique de sécurité à l'aide des moyens offerts par la technique de clé publique. Cette identification est nécessaire pour lier des droits à un agent mobile identifié. Pour cela les droits et les permissions doivent être définis par l'hôte avant de recevoir un agent mobile. Le contrôle d'accès applique les droits et les restrictions avant l'exécution du code mobile afin de prévenir l'accès illégal aux ressources. La confidentialité est satisfaite si une ressource n'est accessible que par les agents autorisés. Cette politique de contrôle d'accès est un raffinement d'une politique de bac à sable par une politique spécifique à chaque application ou classe d'agents mobiles. En fonction des agents, l'hôte peut autoriser ou non l'accès à un ensemble précis de fonctionnalités ou de ressources. Le contrôle d'accès permet de combiner les deux premières techniques en offrant aux agents signés plus de fonctionnalités qu'un simple bac à sable sans pour autant accéder à toutes les fonctionnalités. En contrepartie, l'application d'un tel schéma d'accès a un coût puisque la négociation qui en découle est effectuée dynamiquement à l'exécution.

4.3 Une architecture logicielle basée sur les « *spaces* »

De nombreux modèles de conception et d'architectures logicielles ont été développés pour l'exécution d'applications parallèles sur des grilles de calcul (cf. Section 1.1). L'architecture proposée par notre framework *MCA* utilise les « *spaces* », notamment pour les composants *ComputationSpace* et *Workspace* (cf. Figure 4.1). Dans notre cas, ce type de composant facilite le partage de données dans le but de résoudre un cas de calcul. L'utilisation des *spaces* offre le modèle dit « *ferme de travailleurs* » (spécifié à la section 2.2.2.5 par l'équation 2.37). Celui-ci étend le paradigme Maître-Travailleurs, présenté dans la section 1.1, afin de l'adapter à un système distribué dans un environnement hétérogène. Dans cette section, nous commençons par définir la notion de « *space* » (Section 4.3.1) puis nous définissons le modèle « *ferme de travailleurs* » (Section 4.3.2) sur lequel se base l'architecture logicielle proposée par notre framework MCA. Enfin nous présentons les différents mécanismes rendant une application basée sur les *spaces* tolérante aux pannes (Section 4.3.3).

4.3.1 Définition d'un « *space* »

Au début des années 80, le professeur Gelernter mis les premières briques à la construction d'une architecture basée sur un « *space* » lorsqu'il développa le langage de programmation *Linda* [Gel93] dans le but de faciliter le développement d'applications distribuées. *Linda* était composé d'un ensemble réduit d'opérations combiné avec une mémoire globale, le fameux « *tuple-space* », permettant le stockage de « *tuples* ».

Ce type d'architecture fournit un modèle de développement simple qui remplace complètement le paradigme *RPC* (*R*emote *P*rocedure *C*all) [Mic88]. Le nombre d'opérations qu'il propose à travers un « *space* » est infime mais permet à une large gamme d'applications de profiter de ses avantages comme la modularité, l'évolutivité ou encore la simplicité du code source.

Un *space* est une mémoire virtuelle distribuée et partagée. L'interface de programmation destinée à utiliser un *space* doit contenir au minimum les quatre opérations suivantes : *write*, *read*, *take* et *notify*. Les opérations *write* et *read* permettent respectivement de stocker ou de lire des objets (appelés *entrées*) dans un « *space* » alors que l'opération *take* lit puis supprime une *entrée* de cette mémoire. Enfin, l'opération *notify* permet de s'enregistrer pour être notifié de l'activité du *space*.

Si le modèle RPC fait communiquer directement des objets entre eux à travers des appels explicites de méthodes, l'utilisation d'un *space* implique que seul ce dernier communique avec tous les autres participants. Il est possible de schématiser cela par une sorte de communication par tableau noir [10] où les participants n'ont aucune connaissance des autres participants. Dans ce type d'architecture chaque participant est indépendant des autres. Un *space* fournit un ensemble de caractéristiques permettant de mettre en place des systèmes distribués performants. Nous listons par la suite ces différents caractéristiques :

- *Les spaces sont des mémoires partagées* - De nombreux processus distants peuvent interagir simultanément avec un *space*. Un *space* gère lui-même les accès concurrents ce qui permet aux développeurs de se concentrer sur les données et non sur les accès.
- *Les spaces sont des mémoires dites « associatives »*- La recherche d'un objet dans un *space* se fait par association. Cela fournit un moyen très simple pour trouver un objet : il suffit de définir un modèle (*template*) avec les informations connues de l'*entrée* recherchée pour la retrouver dans un *space*.
- *Les spaces sont tolérants aux pannes* - Nous reviendrons sur ce point dans la section 4.3.3.
- *Les spaces permettent d'échanger du code exécutable* - Dans un *space* les *entrées* sont des données passives (impossible de les modifier ou d'invoquer une de leurs méthodes) mais une fois les objets récupérés en local (via les opérations *read* et *take*), il est possible de les modifier ou d'exécuter leurs méthodes.

10. [EHRLR80] enrichit le concept d'échange d'information en élaborant l'idée du tableau noir (*blackboard* en anglais) avec le projet *HERSAY-II*. Le tableau noir est alors défini comme une zone de travail commune, dévolue à la transmission d'information entre les différents agents

JavaSpaces [FHA99] est une spécification qui définit un service fournissant un mécanisme d'échange et de coordination distribuée pour des objets Java. Il reprend les principes du langage de programmation *Linda* vu précédemment. La spécification *JavaSpaces* fait partie de la technologie Jini™ . Le service éponyme propose une interface simple pour mettre en place des *spaces* et une API pour interagir avec eux. *Outrigger* est le nom donné au service par l'implémentation de référence de la spécification *JavaSpaces* et développée par *Sun*. Cette solution est livrée avec le *Jini starter kit* fourni par *Sun*. Il existe d'autres implémentations disponibles comme *Blitz* [Bli], projet *open source*, écrit et maintenu par Dan Creswell. Il paraît difficile de ne pas citer la plate-forme *GigaSpaces* [Gig11] tant elle propose une solution complète et très bien documentée [Gig]. A la différence des deux premières implémentations, celle-ci n'est pas libre mais propose une version allégée et gratuite.

Nous présentons dans la section suivante comment l'utilisation des *spaces* nous permet de définir un modèle d'architecture logicielle pour l'exécution d'une application distribuée autonomique (cf. Section 1.2) comme le propose notre framework *MCA*.

4.3.2 Implantation d'une « ferme de travailleurs »

Le paradigme *Maître-Travailleurs*, présenté dans la section 1.1.3, offre une solution performante pour les problèmes qui ont un ratio calcul-communication important : c'est à dire lorsque le temps de calcul réel des *travailleurs* dépasse de loin le temps pris pour communiquer avec le reste du système. Ce ratio est directement influencé par la taille d'une tâche relativement à la taille du problème complet auquel elle appartient. En effet, il est possible de diviser un calcul en un nombre important de petites tâches dont le temps de calcul est très court, mais les communications nécessaires entre toutes les tâches du calcul sont plus consommatrices de temps que le calcul réel de toutes les tâches. D'un autre coté, le partage d'un calcul en un petit nombre de grandes tâches peut entrainer une sous-utilisation des ressources de calcul avec un nombre de tâches plus petit que le nombre de *travailleurs* disponibles.

La problématique est alors de savoir comment rendre ce modèle capable de s'adapter aux changements (cf. Section 1.2) au cours de l'exécution d'un calcul. En effet, il est intéressant de pouvoir augmenter le nombre de *travailleurs*, pour répondre à une augmentation de la demande de calcul, ou de réduire le nombre de *travailleurs* si ces derniers ne sont pas utilisés. Comment ajouter ou diminuer le nombre de *travailleurs* de manière autonome, sans avoir à arrêter, reconfigurer et redémarrer le système ? Dans un système où la demande de ressources de calcul peut être variable, comment faire pour automatiser le passage à l'échelle ? Comment automatiser la variation de ressources de calcul en fonction de la demande de calcul sans une intervention humaine ? La suite de cette section montre comment l'utilisation des *spaces* répond à ces questions.

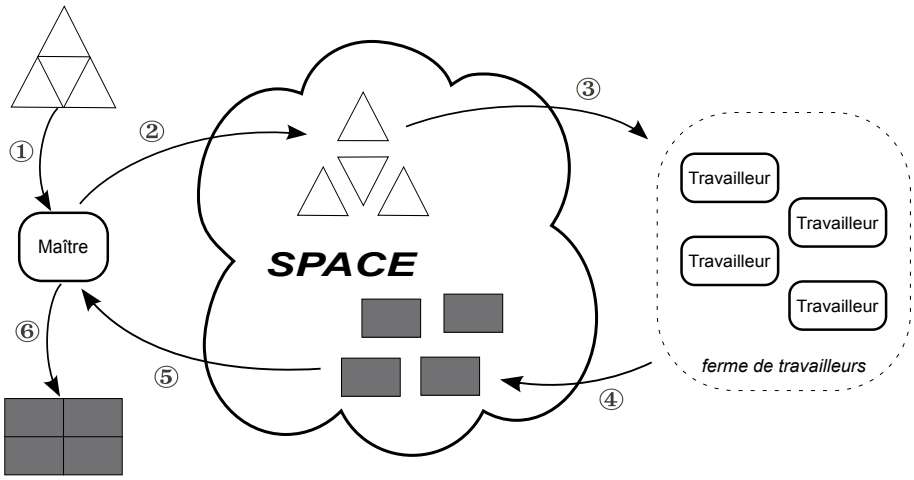


FIGURE 4.11 – Le paradigme « ferme de travailleurs » : ① Le Maître partage le calcul en plusieurs tâches. ② Il dépose les tâches dans le *space* ③ Les travailleurs récupèrent les tâches depuis le *space*. ④ Ils déposent ensuite les résultats dans le *space*. ⑤ Le Maître lit les résultats. ⑥ Pour finir il assemble tous les résultats pour finaliser le calcul.

4.3.2.1 Une équilibrage de charge autonomique

L'utilisation d'un *space* permet de mettre en œuvre le modèle *ferme de travailleurs*. Celui-ci définit un processus *maître* qui écrit des unités de calcul (les tâches) dans un *space*. Celles-ci sont récupérées, exécutées et les résultats retournés dans le *space* par des *travailleurs*, comme nous pouvons le voir dans la figure 4.11.

L'utilisation d'un *space* rend le modèle *Maître-Travailleurs* dynamique et adaptable à la demande de calcul [EG02]. En effet, le processus *maître* ne communique pas directement avec les processus *travailleurs*. Il ne connaît ni le nombre ni la localité des *travailleurs* disponibles car il n'assigne pas les tâches aux travailleurs comme dans le paradigme Maître-Travailleurs classique.

Pour une efficacité optimale, le traitement des tâches doit être équilibré entre les différents *travailleurs* afin que certains ne soient pas surchargés de travail alors que d'autres restent inactifs. Généralement des techniques d'équilibrage de charge sont utilisées pour distribuer la charge de calcul sur les différents *travailleurs* de manière à atteindre ce but. Malheureusement, les grilles de calcul sont le plus souvent composées de machines plus ou moins performantes ce qui rend encore plus difficile un équilibrage de charge efficace.

Les *spaces* apportent une solution simple à ce problème. Chaque *travailleur* obtient une tâche en effectuant une opération *take* sur le *space*. Cette opération facilite l'équilibrage de charge car le flux des tâches vers les *travailleurs* est régulé par les *travailleurs* eux-mêmes. Les *travailleurs* récupèrent les tâches disponibles dans le *space* uniquement quand ils sont prêts. La charge est alors équilibrée par le système lui-même. C'est le *space* qui garantit que chaque tâche n'est prise qu'une seule fois. Ainsi, les *travailleurs* les plus rapides prennent plus de tâches que les *travailleurs* plus lents.

4.3.2.2 Des travailleurs adaptables

Un axe important dans la recherche de l'optimisation des ressources disponibles est le temps d'inactivité des *travailleurs*. Un *travailleur* inactif est une ressource disponible non utilisée. Afin de garantir une activité optimale des *travailleurs*, il est important d'assurer qu'ils soient capables d'exécuter tout type de tâche. Ainsi, tant qu'il y a des tâches à exécuter, un *travailleur* n'est jamais inactif car il est capable d'exécuter toutes les tâches sans exception.

L'utilisation du protocole de découverte offert par Jini™ (cf. Section 4.2.1) permet aux *travailleurs* d'utiliser un *space* sans connaître sa localité sur le réseau. De cette façon, un *travailleur* nouvellement ajouté au système peut trouver le *space*, quel que soit l'emplacement du *travailleur* et du *space* sur le réseau.

L'architecture définie par notre framework *MCA* se base sur le modèle *ferme de travailleurs*. Notre framework propose un environnement d'exécution dans lequel la résolution d'un cas de calcul donne lieu à l'exécution d'un nouveau *space* représenté par le composant *ComputationSpace*. Chaque cas de calcul a un nombre non défini de *travailleurs* qui peut évoluer durant la résolution du cas de calcul. Ces *travailleurs*, représentés par le composant *MCAWorker*, sont tous identiques et sont capables d'exécuter toutes les tâches se trouvant dans un *space* indépendamment des cas de calcul auxquels ils participent. De plus, notre architecture ne définit pas de composant pour représenté un processus *maître* mais donne la possibilité de définir différents types de tâche qui sont exécutées par les *MCAWorker*. Il est donc possible de définir une tâche qui modélise un processus *maître*, celle-ci est traitée par un *MCAWorker* comme toutes les autres tâches.

4.3.3 Tolérance aux pannes dans les « *spaces* »

Nous avons vu précédemment qu'un composant de type *ComputationSpace* est un *space* dédié à la résolution d'un cas de calcul (cf. Section 4.1). Les agents de type *MCAWorker* partagent et modifient les *entrées* contenues dans un *ComputationSpace* afin de résoudre le cas de calcul associé. De ce fait, le *ComputationSpace* est un élément central de l'architecture proposée par notre framework MCA. Celui-ci doit donc être tolérant aux pannes [11], c'est pourquoi nous présentons dans cette section les différents mécanismes offerts par les *spaces* pour répondre à une défaillance des différents *spaces* présents lors de la résolution d'un cas de calcul. Nous abordons les notions de transactions distribuées, de réplication et de persistance de données.

4.3.3.1 Les transactions distribuées

Les transactions distribuées sont présentes dans la plupart des systèmes distribués. Appliquées à une architecture basée sur les *spaces*, comme celle que nous proposons avec le framework *MCA*,

11. Notons que le composant de type *Workspace* est aussi un *space* et jouit donc des mêmes mécanismes de tolérance aux pannes.

les transactions distribuées permettent de réaliser une suite d'opérations avec un *space* de manière transactionnelle. Par exemple, les différentes *entrées* contenues dans un *ComputationSpace* forment un ensemble cohérent dans le but de résoudre un cas de calcul. Si l'ensemble de ces entrées venait à être incohérent, cela pourrait rendre le *ComputationSpace* instable et mettre en péril la résolution du cas de calcul associé.

L'utilisation de transaction lors de l'exécution d'une suite d'opérations sur un *space* fournit alors un moyen de respecter l'intégrité des *entrées* contenues dans le *space*. La technologie *JavaSpaces* utilise le modèle de transaction offert par la technologie Jini™ . Celle-ci propose une solution très simple : toutes les transactions sont supervisées par un (ou plusieurs) service nommé « *Transaction Manager* ». Le *Jini starter kit* fournit une implémentation de ce service, nommée *Mahalo*. Une application qui souhaite exécuter des opérations dans un contexte transactionnel demande au service *Mahalo* de créer une nouvelle transaction. Cette transaction est alors utilisée lors de toutes les opérations. Une transaction peut se terminer de deux manières : par un succès (via la méthode *commit*) et dans ce cas toutes les opérations exécutées sous cette transaction sont effectives, ou par un échec (via l'opération *cancel*) et dans ce cas toues les opérations sont annulées.

Les transactions utilisées dans un environnement *JavaSpaces* observent les propriétés *ACID* (*A*tomicité, *C*ohérence, *I*solation et *D*urabilité) comme toutes transactions, c'est à dire :

- *Atomicité* : toutes les modifications réalisées sous une même transaction sont atomiques : soit toutes les modifications sont effectuées soit aucune d'entre elles. Une modification est l'effet de l'invocation des opérations *read*, *write*, *take* ou *notify*.
- *Cohérence* : toutes les modifications réalisées sous une même transaction ne doivent pas violer la cohérence des *entrées* contenues dans le *space*. Par exemple, l'unicité d'une tâche doit être respectée dans un *ComputationSpace*.
- *Isolation* : de nombreuses transactions peuvent s'exécuter simultanément, il est donc important que chaque transaction travaille dans un mode isolé. Par exemple, deux transactions ne doivent pas se perturber et les modifications faites dans un *space* sous une des transactions ne sont pas visibles par des opérations réalisées au même moment sous une autre transaction.
- *Durabilité* : la fin d'une transaction implique que le *space* se trouve dans un état stable et durable. Soit les modifications sont validées par l'opération *commit*, soit le *space* retourne dans un l'état stable antérieur avec l'opération *cancel* ou par la fin du bail de la transaction. Cette propriété peut résister à une défaillance partielle du système grâce à la possibilité de rendre les services *Jini* persistants (ici les services *Outrigger* et *Mahalo*).

Une transaction permet de regrouper une opération *take* et une opération *write*. Si la transaction échoue, toutes les *entrées* récupérées sous cette transaction (via l'opération *take*) sont remises dans le *space* à la fin de la transaction et toutes les *entrées* écrites (via l'opération *write*) sont supprimées. Le *space* se retrouve comme si les opérations n'avaient jamais été exécutées. Nous revenons sur l'utilisation des transactions lors du traitement d'une tâche par un agent *MCAWorker* (cf. Section 4.4.2.1).

4.3.3.2 La réplication

Le processus de réplication proposé par notre framework MCA permet de dupliquer les *entrées* d'un *space* « *source* » vers un (ou plusieurs) *space* « *cible* ». Ce processus est là pour répondre aux pannes pouvant arrivées durant l'exécution d'un *space*. Dans ce but, il est possible de regrouper plusieurs *spaces* ensemble afin de leur appliquer une politique de réplication commune. L'ensemble de ces *spaces* forme alors un *groupe de réplication*. Il est possible de définir un *ComputationSpace* comme un groupe de réplication pour le rendre tolérant aux pannes. Dans ce cas, un *ComputationSpace* est toujours vu par les agents *MCAWorker* comme un *space* alors qu'il contient en réalité plusieurs instances de *spaces*. Toutes ces instances suivent alors la même politique de réplication.

Une politique de réplication permet d'organiser les différentes instances de *spaces* au sein d'un groupe de réplication et définir un mode de réplication (*synchrone* ou *asynchrone*) pour la réplication de leurs entrées.

Les topologies

L'organisation des différentes instances de *spaces* au sein d'un groupe de réplication est définie par deux types de topologies : « *Actif - Passif* » ou « *Actif - Actif* ».

La topologie « *Actif - Passif* » demande de définir une instance dite « *active* ». Les interactions avec le groupe passent par cette instance *active* et toutes les autres instances dites « *passives* » sont des sauvegardes de l'instance active. Chaque instance passive est la copie exacte de l'instance active. Si l'instance active échoue, une des instances passives est élue par les autres comme la nouvelle instance active.

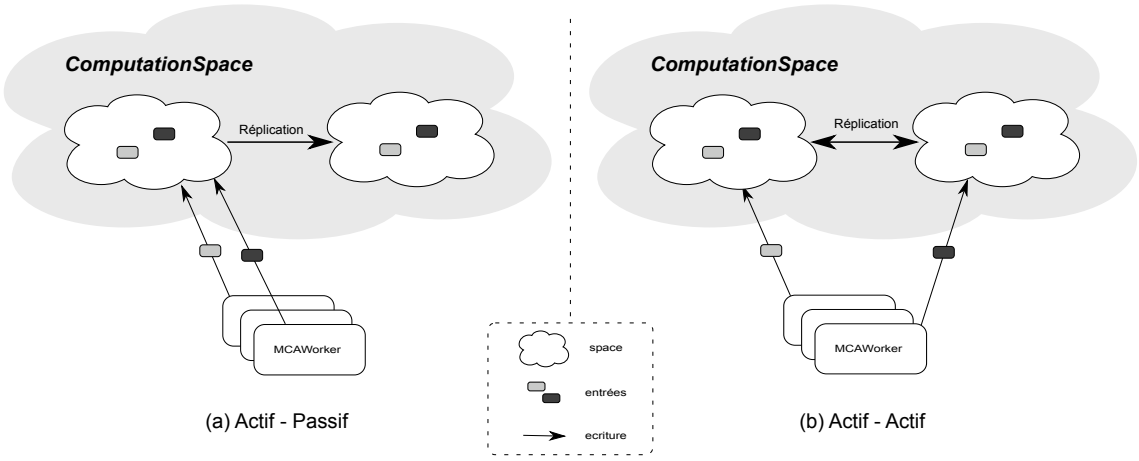


FIGURE 4.12 – Les deux types de topologies possibles au sein d'un *ComputationSpace* défini comme un groupe de réplication.

La topologie « *Actif - Actif* » définit toutes les instances d'un groupe de réplication comme des instances actives et les clients peuvent interagir avec toutes les instances. Chaque instance

réplique ses *entrées* sur toutes les autres instances, s'assurant ainsi que toutes les instances contiennent le même ensemble d'entrées. A son arrivée dans le groupe, une nouvelle instance recherche une instance active pour récupérer ses entrées avant de devenir elle-même active et disponible.

Les modes de réplication

La réplication des *entrées* d'une instance de *space* vers une autre instance à l'intérieur d'un même groupe de réplication peut être définie selon deux modes :

Une *réplication synchrone* assure que le client recevra la réponse de son opération sur l'instance source uniquement lorsque tous les autres instances du groupe de réplication auront exécuté cette opération. Ce mode de réplication est plus adapté à la topologie « *Actif - Passif* » lorsque l'application a besoin de garantir qu'aucune opération exécutée sur l'instance source ne sera perdue et non exécutée sur les autres instances du groupe. En contrepartie, ce type de réplication pénalise le niveau de performance des différentes opérations.

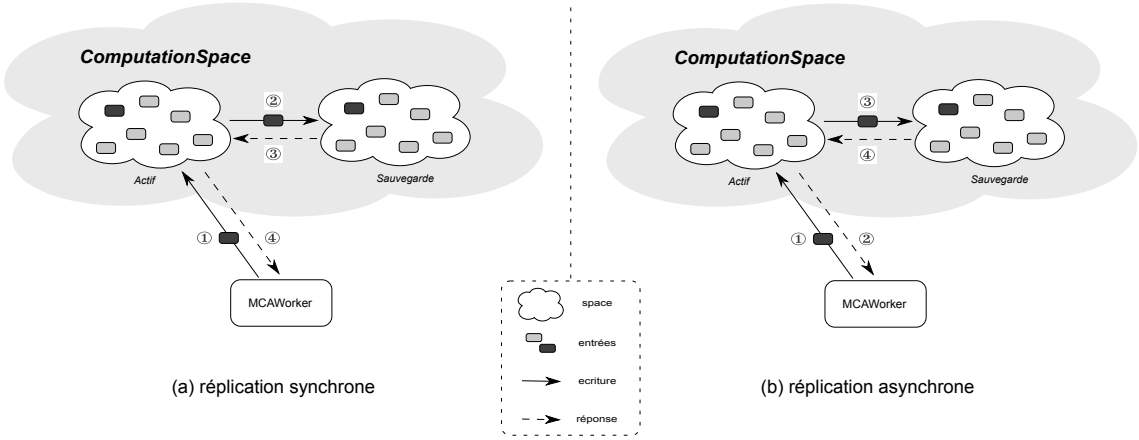


FIGURE 4.13 – Les deux modes de réplication possible au sein d'un *ComputationSpace* défini comme un groupe de réplication. Lors d'une réplication synchrone (a), l'agent *MCAWorker* est bloqué tant que le space initial (*Actif*) n'a pas répliqué la donnée sur le space de sauvegarde (*Sauvegarde*) alors que la réplication asynchrone (b) permet à l'agent *MCAWorker* de continuer son exécution avant la réplication.

Lors d'une *réplication asynchrone*, les opérations sont exécutées sur l'instance source et la réponse est immédiatement renvoyée au client. Toutes les opérations sont accumulées dans l'instance source et sont envoyées de manière asynchrone à l'instance cible, après une période de temps définie ou après un nombre défini d'opérations. Ce type de réplication offre de meilleures performances par rapport à une réplication synchrone mais des *entrées* peuvent être perdues si une panne de l'instance source survient lors de l'envoi des opérations en attente vers l'instance cible. Ce mode de réplication pose un autre problème : la cohérence des *entrées* contenues dans les instances *source* et *cible* n'est pas toujours assurée.

Propriété	Réplication synchrone	Réplication asynchrone
Perte de donnée	Aucune perte de données	Pertes de données possibles lors de la panne d'une instance source avant la réplication des entrées modifiées lors des dernières opérations.
Latence du réseau	Peu tolérant à une haute latence du réseau	Très tolérant à la latence réseau. À utiliser lorsque les instances sont situées dans différents sites géographiques.
Performance	Le client doit attendre la confirmation des instances source et cible. La performance est dépendante des ressources (CPU/Mémoire) des instances source et cible ainsi que de la qualité du réseau entre ces instances.	Le client reçoit une confirmation immédiatement après l'exécution de l'opération sur l'instance source. La performance est uniquement dépendante des ressources (CPU/Mémoire) de l'instance source.
Intégrité des données	Très précise	Peu précise

TABLE 4.2 – Tableau comparatif des deux modes de réplication.

Dans les deux modes de réplication, si une instance cible n'est pas disponible lors d'une opération sur l'instance source, le client reçoit une réponse de l'instance source. L'opération est réalisée sur l'instance cible uniquement lorsque l'instance source rétablit la connexion avec l'instance cible. L'instance source garde la liste de toutes les opérations non-répliquées jusqu'à ce qu'il soit en mesure de rétablir une connexion avec l'instance cible. Le tableau 4.2 présente une comparaison entre la réplication synchrone et asynchrone.

Lorsque la réplication est activée pour un *ComputationSpace*, l'ensemble des *spaces* qui le composent sont organisés en suivant une topologie de type « *Actif - Passif* » en utilisant un mode de réplication asynchrone. Il est tout à fait possible de configurer le *ComputationSpace* pour qu'il utilise une autre topologie et un autre mode de réplication. La section 5.3.2 évalue la performance d'un *ComputationSpace* en fonction des différentes configurations de réplication pour mettre en évidence les avantages et inconvénients de chacune d'entre elles.

4.3.3.3 La persistance de données

Une instance de *space* est une mémoire volatile car les *entrées* qu'elle contient sont perdues à la fin de son exécution. Pour permettre à une instance de *space* de recharger ses *entrées* après un redémarrage, notre framework *MCA* donne la possibilité de sauvegarder ses *entrées* dans une source de données externe. Nous avons choisi la solution offerte par les bases de type *NoSQL* (comprendre « *Not Only SQL* ») [Tiw11]. Cette solution est adaptée au stockage des *entrées* d'un *space* car ce type de base est « *schemaless* », c'est à dire qu'il n'est pas nécessaire de définir un schéma pour stocker des données. *MongoDB* [DC10] propose une base *NoSQL* orientée document

et stocke les données au format *BSON*[12] [Bso] (cf. Figure 4.14). La manipulation des données dans ce format est simple avec l'aide d'*API* fournies pour le langage Java.

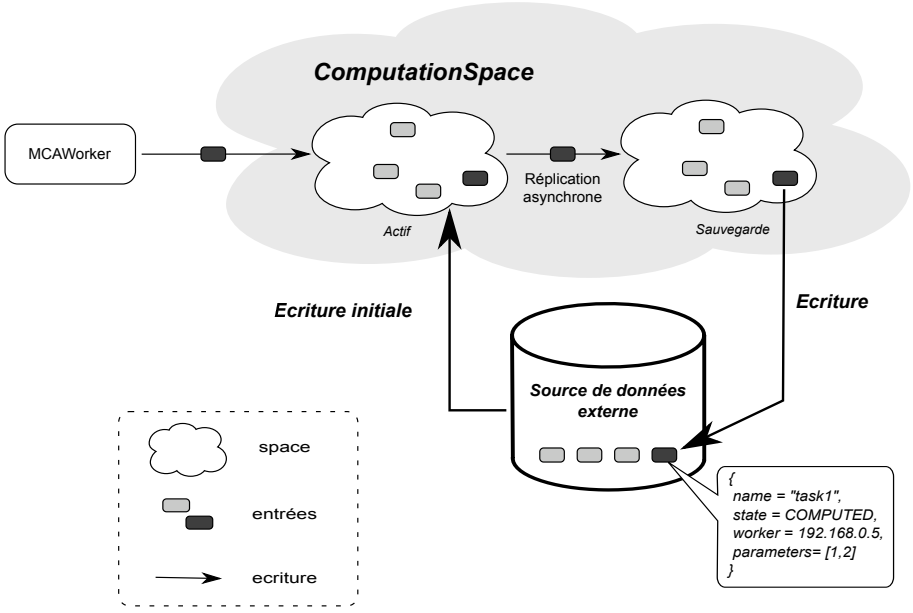


FIGURE 4.14 – La persistance des données d'un *ComputationSpace*.

Rendre une instance de *space* persistante par l'utilisation de cette persistance des données entraine une baisse de performance. Il est possible de réaliser une persistance dite « asynchrone ». Le même mécanisme que celui présenté pour la réplication asynchrone (cf. Figure 4.13) est utilisé. Dans ce cas, l'instance persistante redonne la main au client avant de sauvegarder ses entrées dans la source de données externe. La section 5.3.2 évalue la performance d'un *ComputationSpace* en fonction de l'activation ou non de la persistance de ses données.

4.4 Le framework MCA

Nous présentons dans cette section les différents composants présents au sein de l'architecture définie par le framework MCA. Les exigences imposées à ces composants ont été spécifiées au chapitre 2. Des propriétés temporelles ont été établies au chapitre 3 à propos de la disponibilité des agents *MCAWorker* et de l'évaluation des tâches de calcul (cf. Section 3.3.2). Nous commençons par décrire la plate-forme de calcul (Section 4.4.1) à laquelle se connectent les agents *MCAWorker* (Section 4.4.2) pour participer à la résolution de cas de calcul. Ensuite, nous abordons l'utilisation des agents mobiles de type *ComputeAgent* (Section 4.4.3) qui contiennent le code des différentes tâches d'un cas de calcul. Pour finir, nous présentons la définition et l'utilisation des structures de données distribuées (Section 4.4.4) proposées par notre framework MCA.

12. *BSON* pour *Binary JSON*, *JSON* (pour *JavaScript Object Notation*) étant un format de données dérivé de la notation des objets du langage *JavaScript*. L'avantage de ce format est qu'il est assez abstrait et générique pour pouvoir représenter n'importe quel type de données.

4.4.1 La plate-forme de calcul

Les composants *MCAService* et *ComputationSpace* constituent la plate-forme de calcul qui est l'élément central de l'architecture définie par notre framework MCA.

4.4.1.1 Le service *MCAService*

Le framework *MCA* utilise la technologie Jini™ (cf. Section 4.2.1) pour faire communiquer les différents composants de l'architecture. Il fournit le service *MCAService* (spécifié dans le chapitre 2 par le terme *CaseDirectory* (eq. 2.61)) qui est le point d'entrée des agents *MCAWorker* (cf. Section 4.4.2) pour se connecter à notre plate-forme de calcul. Le service *MCAService*, qui expose l'interface de la figure 4.15, est développé comme un service *Jini* (cf. Section 4.2.1). L'invocation de ses méthodes par les agents *MCAWorker* est réalisée via un *proxy* enregistré dans un annuaire. Les agents *MCAWorker* disposent des différents protocoles de découverte présentés par la figure 4.3 afin de trouver l'annuaire et de récupérer le *proxy* du service *MCAService*. L'utilisation du proxy respecte la politique de sécurité définie dans la section 4.2.1.4.

```
1 public interface MCAService extends Remote {
2   public ComputationCase addCase(String name, String description) throws MCASpaceException;
3   public ComputationCase addCase(String name, String description,
4             RecoveryTaskStrategy strategy) throws MCASpaceException;
5   public ComputationCase getCase(String name) throws MCASpaceException;
6   public Collection<ComputationCase> getCases() throws MCASpaceException;
7   public void removeCase(String name) throws MCASpaceException;
8   public EventRegistration register(MCASpaceEventListener listener, long leaseTime) throws MCASpaceException;
9 }
```

FIGURE 4.15 – Interface du service *MCAService*.

L'invocation d'une des méthodes `addCase` ajoute un cas de calcul à la plate-forme de calcul. Il est possible de définir une stratégie ou de laisser celle définie par défaut (cf. Section 4.4.1.2 avec les *entrées* de type *RecoveryTaskStrategy*). Cet appel entraine la création d'un nouveau *ComputationSpace* dédié spécialement à la résolution d'un cas de calcul. Ce *ComputationSpace* n'est pas accessible directement mais via un agent de type *ComputationCase* qui permet de communiquer avec lui. Ce composant, comparable à un *proxy*, est défini par l'interface `ComputationCase` et propose des méthodes permettant de réaliser des opérations sur le *ComputationSpace* associé. La figure 4.16 présente les échanges entre un agent *MCAWorker* et un *ComputationSpace*. Un agent *MCAWorker* obtient alors un agent *ComputationCase* via la méthode `getCase`. Il peut alors effectuer des opérations sur le *ComputationSpace*. Si la méthode `getCase` ne retourne aucun agent *ComputationCase*, l'agent *MCAWorker* peut demander au service *MCAService* d'être prévenu si un cas de calcul venait à manquer de ressources via la méthode `register`. Dans ce cas, l'agent

MCAWorker s'ajoute au pool d'agents *MCAWorker* disponibles (cf. Figure 4.2).

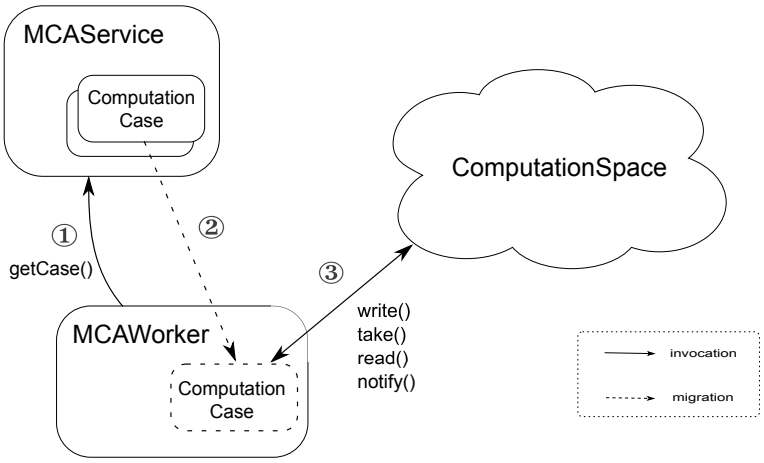


FIGURE 4.16 – Communications entre un agent *MCAWorker* et le service *MCAService*. ① L'agent *MCAWorker* demande au service *MCAService* si un cas de calcul a besoin de ressource. ② Un agent mobile *ComputationCase* migre vers l'agent *MCAWorker*. ③ L'agent *MCAWorker* peut maintenant communiquer avec le *ComputationSpace* via l'agent *ComputationCase*.

4.4.1.2 Le ComputationSpace : un *space* particulier

La particularité d'un *ComputationSpace*, par rapport à un *space* classique (cf. Section 4.3.1), se situe dans les types d'*entrées* qu'il peut recevoir. La figure 4.17 fournit le diagramme de classe des types d'*entrées* pouvant être écrites dans un *ComputationSpace*. Nous remarquons que ces classes ont toutes le stéréotype *« entry »* et leurs attributs sont tous publiques comme l'impose la spécification *JavaSpaces*. Nous listons ces types d'*entrées* :

State - Il n'y a qu'une seule entrée de ce type par *ComputationSpace*. L'état du cas de calcul (cf. Figure 4.18) est une donnée primordiale car chaque agent *MCAWorker* participant au cas de calcul associé au *ComputationSpace* surveille les modifications de cet état pour continuer ou non la résolution du cas de calcul.

Property - La résolution d'un cas de calcul impose, le plus souvent, la déclaration de propriétés (spécifiées à la section 2.2.3.4 par l'équation 2.48). Celles-ci sont des valeurs partagées par toutes les tâches. Ce type d'*entrée* possède deux attributs : une chaîne de caractères `name` pour identifier la propriété et `value` (instance de type `java.io.Serializable`) pour sa valeur.

Task - Définit une tâche du cas de calcul (spécifiée à la section 2.2.3.2 par l'équation 2.42). Une tâche est caractérisée par les propriétés suivantes :

- `name` : ce nom doit être unique pour un cas de calcul et permet d'identifier chaque tâche du cas de calcul. Cette information est utile, par exemple, lors de la recherche d'un résultat.
- `state` : l'état de la tâche (cf. Figure 4.19) permet, par exemple, aux agents *MCAWorker* de connaître les tâches en attente de traitement (IN_PROGRESS) pour les récupérer.

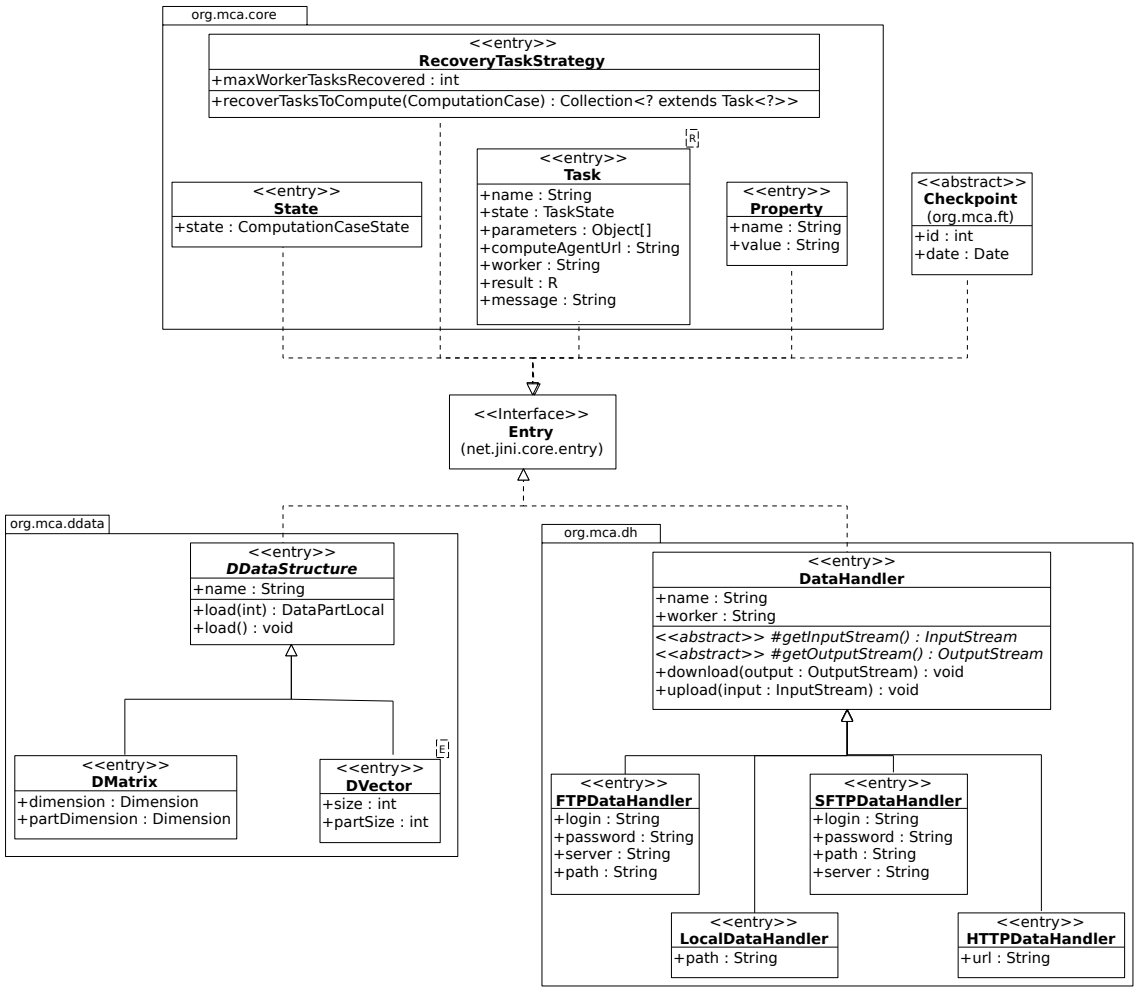


FIGURE 4.17 – Diagramme de classes des types d'*entrées* d'un *ComputationSpace*.

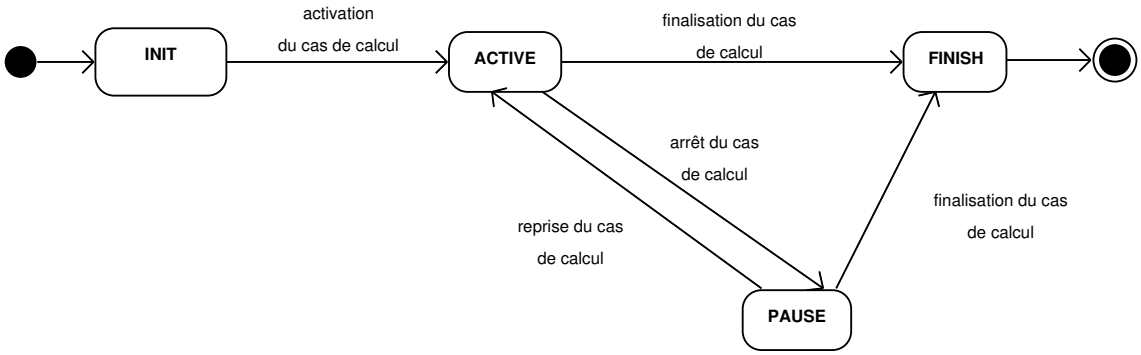


FIGURE 4.18 – Diagramme d'états-transitions d'un cas de calcul.

– `computeAgentUrl` : l'URL de l'agent mobile de type *ComputeAgent* associée à la tâche. Cette information permet à un agent *MCAWorker* de télécharger le code à exécuter correspondant à la tâche (cf. Section 4.4.3).
– `parameters` : à la différence d'une *entrée* de type `Property` qui est une propriété commune à

toutes les tâches du cas de calcul, la liste des paramètres (définie par un tableau) est spécifique à la tâche. Cette liste permet de paramétrer le *ComputeAgent* associé à cette tâche lors de l'exécution du code.

– `result` : le résultat d'une tâche peut être une instance de toute classe implémentant l'interface `java.io.Serializable`. Il peut être, par exemple, de type `DataHandler` si le résultat contient des données de taille importante. Sa valeur peut aussi être *null* si la tâche modifie uniquement des données accessibles par une entrée de type `DataHandler`.
– `message` : ce message décrit l'erreur lorsqu'une tâche se trouve dans l'état FAILURE.
– `worker` : il s'agit de l'adresse de l'agent *MCAWorker* qui traite ou a traité la tâche.

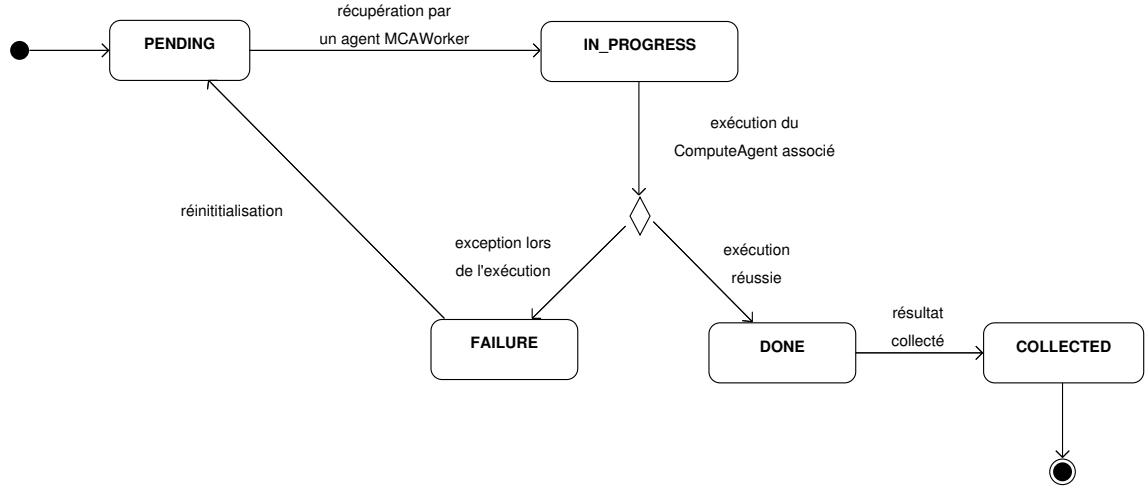


FIGURE 4.19 – Diagramme d'états-transitions d'une *entrée* de type *Task*.

RecoveryTaskStrategy - Il n'y a qu'une seule entrée de ce type par *ComputationSpace*. Celle-ci définit la stratégie utilisée par les agents *MCAWorker* pour récupérer les *entrées Task* à traiter. La stratégie par défaut consiste à récupérer `maxWorkerTasksRecovered` (ayant la valeur 1 par défaut) entrées de type *Task* dont l'état est PENDING. Pour modifier la stratégie par défaut, le développeur doit créer une classe héritant de la classe `RecoveryTaskStrategy` et donner une nouvelle implantation de la méthode `recoverTasksToCompute`. La section 5.3 présente la définition d'un stratégie spécifique à un cas calcul de type *Maître-Travailleurs*.

DataHandler - Les données utilisées pour le traitement des tâches du cas de calcul ne sont donc pas écrites dans le *ComputationSpace* (spécifiée à la section 2.2.3.1 par l'équation 2.39). En effet, ce sont seulement les accès à ces données qui sont présents. La classe abstraite `DataHandler` représente l'accès à des données et propose deux méthodes.

- `public void download(OutputStream output)throws IOException`
- `public void upload(InputStream input)throws IOException`

La méthode `download` permet de télécharger des données dans le flux `output` et la méthode `upload` sauvegarde le flux de données `input` vers la source de données représentée par le *DataHandler*.

Les différentes implantations de cette classe correspondent aux différents types d'accès offerts par

notre framework pour accéder aux données. Ces accès se font par des protocoles de communication (*FTP, SFTP*[13]*, HTTP*) ou directement par le système de fichiers via la classe `LocalDataHandler`. Il est possible de définir d'autres implantations si l'utilisation d'autres protocoles de communication était nécessaire pour l'accès aux données lors de la mise en place d'un cas de calcul.

DDataStructure - Ce type d'*entrée* permet l'utilisation de Structures de Données Distribuées. La mise en place de telles structures est décrite dans la section 4.4.4.

Checkpoint - L'architecture définie par le framework *MCA* est tolérante aux pannes. L'utilisation de points de reprise (*checkpoint*) est une pratique courante dans ce type de système. A partir de ces points de reprise le système doit être capable de reprendre la résolution d'un cas de calcul pour éviter de recommencer le cas de calcul depuis le début. Une entrée de type *Checkpoint* possède au minimum deux attributs : un identifiant et une date. Cette deux informations permettent de choisir un point de reprise lors de la relance d'un cas de calcul. Par défaut, lors de la relance d'un cas de calcul, c'est le point de reprise le plus récent qui est choisi mais il est possible de choisir un point de reprise plus ancien. La définition d'un point de reprise est spécifique à chaque cas de calcul car il contient des informations sur le traitement du cas de calcul lui-même. Pour rendre un cas de calcul tolérant aux pannes, le développeur doit définir un ou plusieurs nouveaux types d'*entrée* héritant de la classe `Checkpoint` spécifiques au cas de calcul. La section 5.2 présente l'utilisation de points de reprise dans un cas de calcul suivant le modèle *SPMD*.

4.4.2 Les agents *MCAWorker*

Les agents de type *MCAWorker* traitent les différentes tâches d'un cas de calcul. Ce type d'agent a été spécifié au chapitre 2 dans la section 2.2.4.2 par le terme $Worker$ (eq. 2.64). L'architecture d'un agent *MCAWorker* s'articule autour de 4 composants : un contexte d'exécution (un *space* de type *WorkSpace*), un moniteur d'activité (composé des processus *MCASpaceListener* et *WorkerMCASpaceEventListener*), un gestionnaire de cas de calcul (processus *CaseStateListener*) et un exécuteur de tâches (processus *TaskExecutor*). Notons que le contexte d'exécution d'un agent *MCAWorker* est actif durant toute l'exécution de l'agent, au contraire des autres composants qui sont eux actifs en fonction de l'état de l'agent. La figure 4.20 présente les différents états d'un agent *MCAWorker* et les différents processus démarrés en conséquence.

Le *contexte d'exécution* d'un agent *MCAWorker* est un *space* défini par le composant *WorkSpace* que nous retrouvons dans le diagramme de la figure 4.1. Ce composant bénéficie des mêmes propriétés de tolérance aux pannes que possède le composant *ComputationSpace*, comme la persistance de données et les transactions (cf. Section 4.3.3). Ce type de *space* se différencie d'un *ComputationSpace* par la définition de sa politique de sécurité. En effet, un *WorkSpace* n'autorise aucune modification provenant d'un processus autre que l'agent *MCAWorker* qui l'a lancé. De plus, le seul accès distant possible en lecture se situe lors de la manipulation de structures de données distribuées (cf. Section 4.4.4). Les types d'*entrée* qui sont acceptés dans un *WorkSpace*

13. *SFTP* pour *SSH File Transfer Protocol*

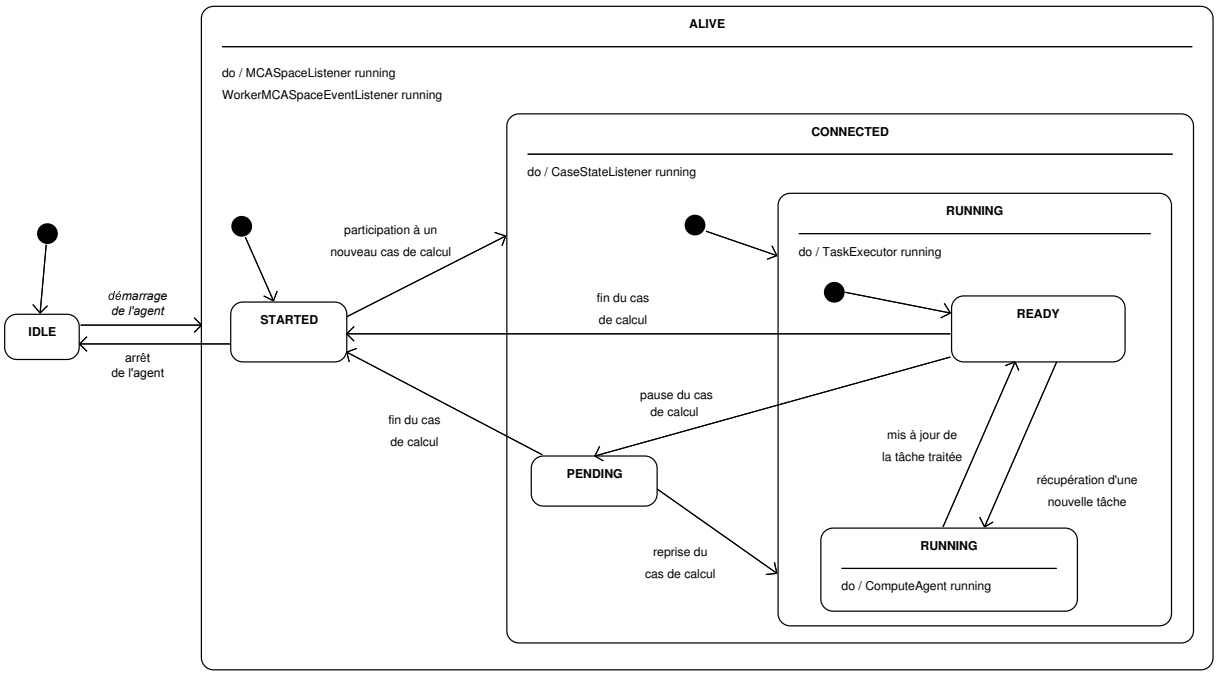


FIGURE 4.20 – Diagramme d'états-transitions d'un agent *MCAWorker*.

sont les types *Task* et *CheckPoint* (cf. Section 4.4.1.2), le type *DData* pour le partage de données avec les autres agents *MCAWorker* (cf. Section 4.4.4) et enfin le type *FTContext* qui contient des informations sur la cas de calcul auquel participe l'agent *MCAWorker*.

Le *moniteur d'activité* surveille l'activité de la plate-forme de calcul. Il se compose de deux processus : le processus *MCASpaceListener* et le processus *WorkerMCASpaceEventListener*. Ces deux processus sont lancés au démarrage de l'agent *MCAWorker*. Le processus *MCASpaceListener* découvre et surveille les différents services *MCAService* disponibles sur le réseau. Le processus *WorkerMCASpaceEventListener* surveille l'activité de chaque service découvert avec l'arrivée ou le départ de cas de calcul. Á l'arrivée d'un nouveau cas de calcul sur la plate-forme, l'agent *MCAWorker* qui ne participe à la résolution d'aucun cas de calcul (état ALIVE de la figure 4.20) est prévenu. Il est alors susceptible de récupérer un proxy de type *ComputationCase* (cf. Figure 4.16). Les informations de ce proxy sont alors sauvegardées dans le *WorkSpace* de l'agent *MCAWorker* (via une *entrée* de type *FTContext*). En cas de panne de l'agent *MCAWorker*, ces informations seront utilisées pour reprendre le cas de calcul auquel il participait lors du redémarrage de l'agent.

Le *gestionnaire de cas de calcul* surveille l'état du cas de calcul (cf. Figure 4.18) auquel l'agent *MCAWorker* participe. Le passage du cas de calcul dans l'état ACTIVE ou PAUSE entraine respectivement l'exécution ou l'arrêt de l'*exécuteur de tâches*. La fin d'un cas de calcul (passage dans l'état FINISH) entraine instantanément l'arrêt de l'exécuteur de tâches, sans attendre le résultat d'une tâche qui serait en cours de traitement. La fin d'un cas de calcul libère l'agent *MCAWorker* pour lui permettre de participer à un autre cas de calcul (état STARTED de la

Figure 4.20).

L'*exécuteur de tâches* récupère les tâches en attente de traitement se trouvant dans le *ComputationSpace*, c'est à dire celles qui sont dans l'état PENDING (cf. Figure 4.19). Pour cela, il utilise la stratégie définie par l'unique *entrée* de type *RecoveryTaskStrategy* qui est disponible dans le *ComputationSpace*. Il exécute les tâches avec la participation du *ComputeAgent* associé à chaque *entrée Task*. Ce processus est démarré par le gestionnaire de cas de calcul lors de la récupération du proxy de type *ComputationCase*. L'exécuteur de tâches est actif tant que le cas de calcul auquel participe l'agent *MCAWorker* est dans l'état ACTIVE. Si ce cas de calcul est mis en pause alors que l'agent est *MCAWorker* en train de traiter une de ses tâches (état RUNNING), ce dernier finit l'exécution de la tâche et sauvegarde le résultat dans son *WorkSpace*. Ainsi lors du redémarrage du cas de calcul, il pourra mettre à jour la tâche traitée sur le *ComputationSpace* du cas de calcul. Nous revenons sur le traitement d'une tâche par un agent *MCAWorker* dans la section suivante.

4.4.2.1 Traitement d'une tâche par un agent *MCAWorker*

Pour traiter une tâche du cas de calcul auquel il participe, un agent *MCAWorker* doit récupérer une *entrée* de type *Task*. Pour cela, il invoque la méthode `getTaskToCompute` (cf. Figure 4.21) sur l'agent *ComputationCase* correspondant (définie par l'interface `ComputationCase`). Cette méthode retire du *ComputationSpace* une *entrée* de type *Task* se trouvant dans l'état PENDING (cf. Figure 4.19). L'agent *MCAWorker* valorise alors les propriétés `state` (PENDING → IN_PROGRESS) et `worker` (avec l'adresse de l'agent *MCAWorker*) de l'*entrée* récupérée et met à jour cette *entrée* dans le *ComputationSpace*.

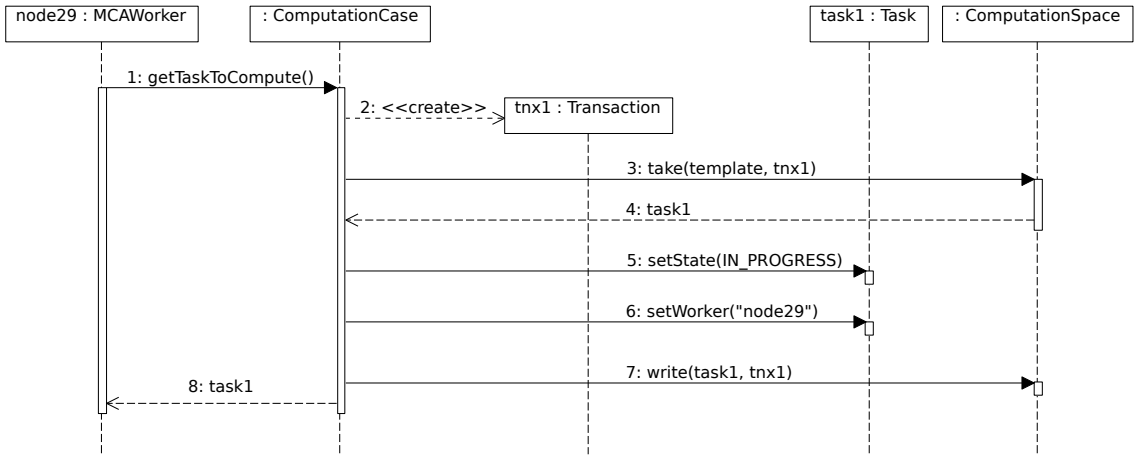
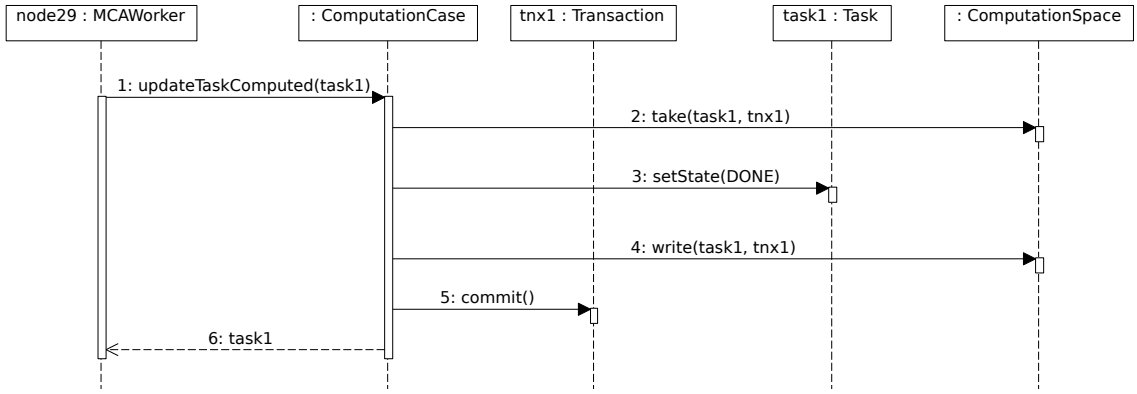


FIGURE 4.21 – Diagramme de séquence de la demande d'une tâche à traiter par un agent *MCAWorker* situé sur l'hôte *node29*. La variable `template` est une instance de la classe `Task` ayant la valeur de son attribut `state` égale à PENDING.

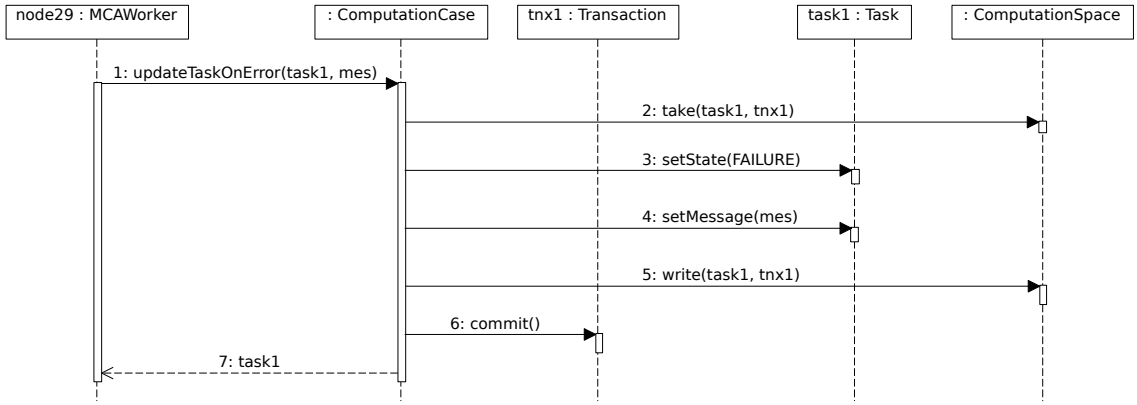
L'*entrée* de type *Task* récupérée est sauvegardée dans le *WorkSpace* de l'agent *MCAWorker*. Le traitement de la tâche correspondante est alors réalisé par le couple *MCAWorker-ComputeAgent*

(cf. Section 4.4.3.1). Deux cas sont alors possibles (cf. Figure 4.22) :

– le code défini par le *ComputeAgent* associé à la tâche s'exécute parfaitement : l'agent *MCAWorker* met à jour l'*entrée* correspondante à la tâche dans le *ComputationSpace* en invoquant la méthode `updateTaskComputed` sur l'agent *ComputationCase* (cf. Figure 4.22a).
– le code défini par le *ComputeAgent* associé à la tâche génère une exception : l'agent *MCAWorker* met à jour l'*entrée* correspondant à la tâche dans le *ComputationSpace* en invoquant la méthode `updateTaskOnError` sur l'agent *ComputationCase* (cf. Figure 4.22b).



(a) exécution réussie



(b) exécution échouée

FIGURE 4.22 – Diagrammes de séquence de la mise à jour d'une tâche par un agent *MCAWorker*.

L'exécution d'une tâche par un agent *MCAWorker* est une opération critique car elle modifie à plusieurs reprises l'*entrée* de type `Task` correspondante dans le *ComputationSpace*. La mise à jour d'une *entrée* est réalisée par deux opérations successives : une opération *take* qui retire l'*entrée* du *space*, et une opération *write* qui écrit l'*entrée* mise à jour dans la *space*.

Une entrée de type *Task* ne doit pas définitivement disparaître du *ComputationSpace* si l'agent *MCAWorker* qui la traite tombe en panne entre les actions *take* et *write*. De plus, elle ne doit pas être bloquée dans l'état IN_PROGRESS (cf. figure 4.19) si l'agent *MCAWorker* qui l'a récupéré pour la traiter tombe en panne. L'utilisation des transactions est donc indispensable pour assurer que cette suite d'opérations « take-write » soit réalisée de manière atomique. Sans

cet aspect transactionnel, plusieurs scénarios pourraient conduire un cas de calcul à être bloqué indéfiniment dans un état non final, c'est à dire dans un état différent de l'état FINISH de la figure 4.18.

4.4.3 L'agent mobile *ComputeAgent*

Un agent de type *ComputeAgent* est un agent mobile. Les contraintes imposées à ce type d'agent sont décrites au chapitre 2 dans la section 2.2.3.2 par l'équation 2.41. Cet agent est capable de migrer vers un agent *MCAWorker* en suivant le modèle défini par une migration réactive (cf. Section 4.2.2.1). Un *ComputeAgent* est donc enregistré dans un annuaire, via le service *Lookup* de Jini™ (cf. Section 4.2.1), pour être mis à disposition des agents *MCAWorker*[14]. Toutes les *entrées Task* d'un *ComputationSpace* sont associées à un type de *ComputeAgent* par son attribut `computeAgentUrl`. Cet attribut définit une URL de la forme `jini://<host>:<port>/<name>` avec `jini://<host>:<port>` pour indiquer l'URL d'un service *Lookup* et `name` pour indiquer le nom donné au *ComputeAgent* lors son enregistrement dans le *Lookup*.

Un *ComputeAgent* contient le code exécuté lors du traitement de la tâche auquel il est associé. Le code d'un *ComputeAgent* est implanté indépendamment de son contexte d'exécution (paramètres spécifiques à la tâche, propriétés du cas de calcul courant...). Deux tâches peuvent très bien être associées au même type de *ComputeAgent*. Par exemple, dans le cas d'un calcul suivant le modèle *SPMD* (cf. Section 5.2) le même code doit être exécuté sur différentes parties d'une structure de données. Dans ce cas, un seul type de *ComputeAgent* est implanté et cet agent est associé à toutes les tâches du cas de calcul. La figure 4.23 présente les différentes classes mises en œuvre lors de la définition d'un type de *ComputeAgent*.

Définir un nouveau type de *ComputeAgent* consiste à implanter une sous-classe de la classe `AbstractComputeAgent` (en implantant la méthode `execute`). Différentes sous-classes sont déjà implantées et fournies par notre framework : les classes `SPMDAgent`, `MasterAgent` et `SkeletonAgent`. Ces trois types de *ComputeAgent* offrent la possibilité de définir des cas de calcul basés sur des paradigmes de calcul parallèle comme *SPMD* (`SMPDAgent`), Maître-Travailleurs (`MasterAgent`) et les squelettes algorithmiques (`SkeletonAgent`). Le chapitre 5 utilise ces trois types de *ComputeAgent* pour évaluer l'architecture *MCA* à travers la résolution de différents cas de calcul.

4.4.3.1 Le couple *MCAWorker-ComputeAgent*

Tous les agents *MCAWorker* sont identiques et cela quelque soit le cas de calcul auquel ils participent. En effet, c'est seulement lors l'exécution d'un *ComputeAgent* qu'un agent *MCAWorker* se différencie des autres agents *MCAWorker*. Un *ComputeAgent* est récupéré grâce à la valeur de l'attribut `computeAgentUrl` défini dans l'*entrée Task* (cf. Section 4.4.1.2) associé à la tâche à traiter par l'agent *MCAWorker*. Après la migration de l'agent mobile vers la machine où se trouve

14. La section 5.1.3.1 présente la configuration et le déploiement d'un *ComputeAgent*.

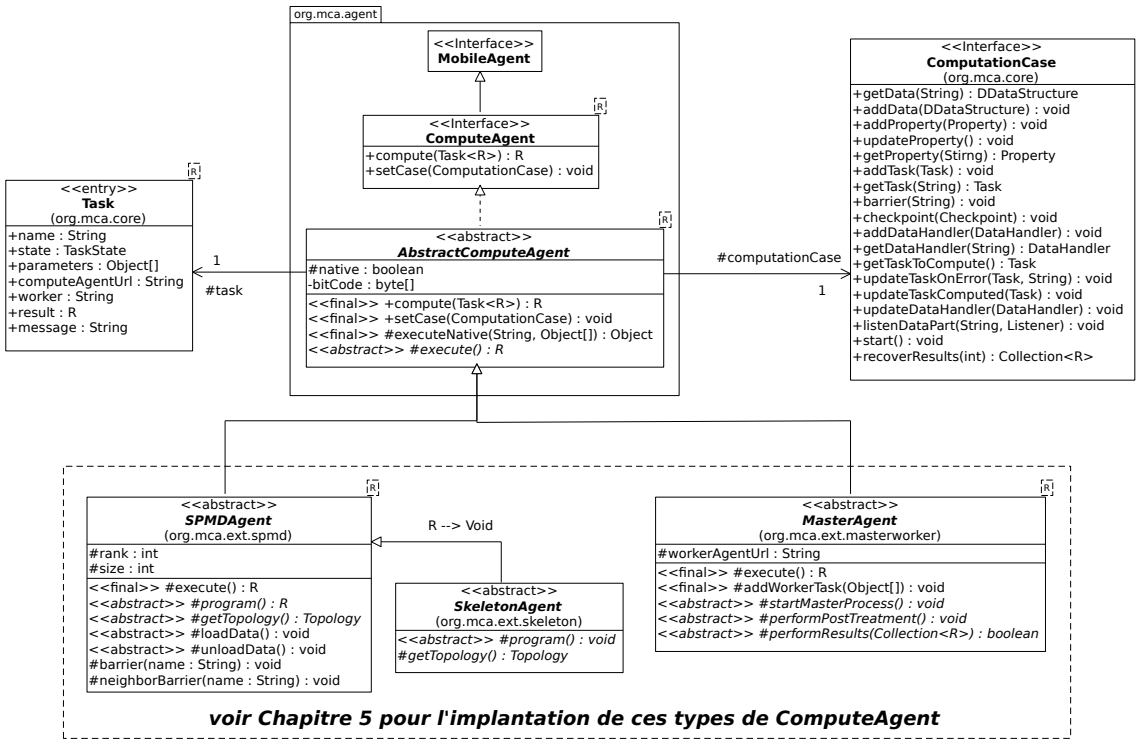


FIGURE 4.23 – Diagramme des classes utilisées pour définir un *ComputeAgent*.

l'agent *MCAWorker*, ce dernier s'assure qu'il peut exécuter le *ComputeAgent* en toute sécurité (cf. section 4.2.2.2).

Pour exécuter le *ComputeAgent*, l'agent *MCAWorker* lui fournit un lien vers le *ComputationSpace* associé au cas de calcul (via l'agent *ComputationCase*) et l'*entrée* de type *Task* correspondant à la tâche qu'il doit traiter. Le *ComputeAgent* exécute alors le code contenu dans la méthode `execute` déclarée dans la classe `AbstractComputeAgent` et implantée dans la classe réelle du *ComputeAgent* récupéré.

4.4.3.2 Exécution de code natif

L'une de propriétés souhaitées pour un outils de simulation numérique est la reprise de code existant (*legacy code*). C'est dans ce sens que le framework *MCA* offre la possibilité de définir un *ComputeAgent* dit « natif ». L'exécution de ce type d'agent ne modifie en rien la relation *MCAWorker-ComputeAgent* vue précédemment. Un agent *MCAWorker* reste identique qu'il exécute un *ComputeAgent* natif ou non. La totalité du code à exécuter doit rester entièrement mobile même si celui-ci est écrit à l'aide d'un langage natif.

Pour répondre à cette contrainte, nous avons choisi d'utiliser la suite d'outils proposée par

LLVM[15]. Elle permet à des fichiers sources écrits à l'aide de langages natifs (*C++, C, Fortran, ADA* ...) d'être compilés dans un format, appelé *bitcode*. Ce format est alors interprété par une machine virtuelle (fournie également par *LLVM*). Lors de la création d'un *ComputeAgent* natif, le code natif à exécuter est compilé en *bitcode* et injecté dans l'agent mobile associé via l'attribut `byteCode` de la classe `AbstractComputeAgent` (cf. Figure 4.23). Cette attribut permet de sauvegarder ce *bitcode* sous la forme d'un tableau d'octets. Ainsi le code natif devient mobile au même titre que le *ComputeAgent* qui le contient. *LLVM* offre la possibilité de manipuler la machine virtuelle qu'il fournit à l'aide d'un ensemble de librairies *C++*. Nous avons ainsi développé une librairie dynamique en *C++* (nommée *libMCA*) qui permet d'exécuter des fonctions définies dans un fichier *bitcode* (cf. Figure 4.24).

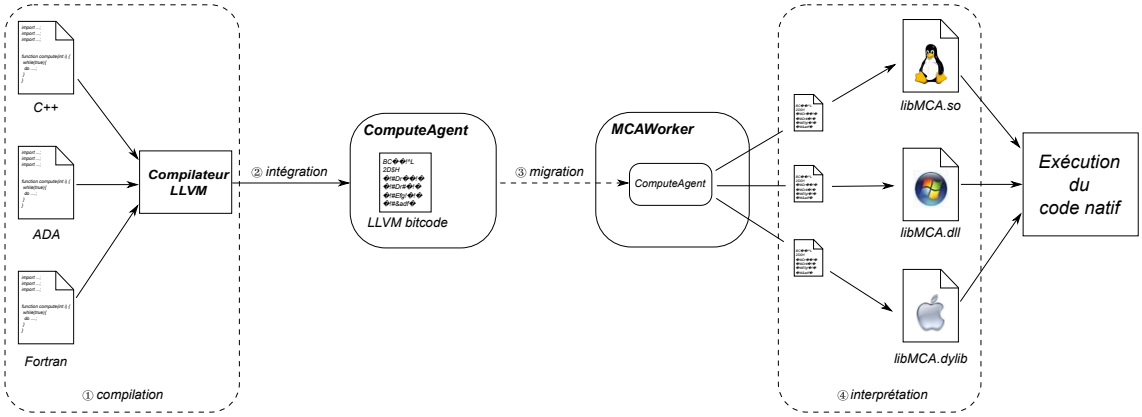


FIGURE 4.24 – Cycle de vie d'un *ComputeAgent* « natif ». ① Le code natif est compilé vers un fichier au format *bitcode*. ② Le *bitcode* est intégré au *ComputeAgent*. ③ Le *ComputeAgent* migre vers l'agent *MCAWorker* voulant exécuter le code. ④ Le *bitcode* contenu par le *ComputeAgent* est interprété grâce à la librairie *libMCA* présente sur la machine sur laquelle s'exécute l'agent *MCAWorker*.

La bibliothèque *libMCA* est installée sur chaque machine où s'exécutent les agents *MCAWorker*. Un *ComputeAgent* « natif » peut invoquer les fonctions proposées par cette librairie et ainsi utiliser les fonctions définies dans le *bitcode* qu'il contient. Nous pouvons citer en particulier les deux fonctions de la librairie *libMCA* suivantes :

- **void** load(**const char**∗ filename);

La première action d'un *ComputeAgent* « natif » est de sauvegarder le *bitcode* qu'il contient dans un fichier dans le système de fichier de la machine sur laquelle il s'exécute. La fonction `load` lui permet alors de charger le fichier sauvegardé en indiquant son emplacement par le paramètre `filename`. Il peut ensuite exécuter le code chargé via la fonction suivante :

- **const char**∗ execute(**const char**∗ **const** function, **const char**∗ **const** parameters[]);

15. *LLVM* (pour *L*ow *L*evel *V*irtual *M*achine) [Lat11] est une boîte à outils pour construire des compilateurs, des éditeurs de liens, des environnements d'exécution, des machines virtuelles et d'autres programmes d'exécution liés à ces outils.

Une fois chargé, le code du fichier bitcode est exécuté via la fonction `execute`. Les paramètres contenus dans le tableau `parameters` sont utilisés pour appeler la fonction `function`.

Un agent de type *ComputeAgent* est écrit en Java comme tous les autres composants de notre architecture. Si celui-ci est « natif », c'est uniquement par le fait qu'il exécute du code écrit dans un langage natif. Ainsi nous avons mis en place une solution afin de pouvoir utiliser notre librairie dynamique écrite en C++ (*libMCA*) à partir du code Java de la classe `AbstractComputeAgent`. Nous avons choisi d'utiliser le framework *JNI*[16] (cf. Figure 4.25) particulièrement adapté à ce type de configuration.

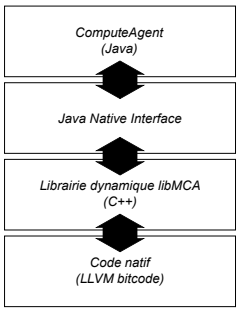


FIGURE 4.25 – Communications par couches lors de l'exécution de code natif.

4.4.4 Les Structures de Données Distribuées

Le dernier type de composant de l'architecture MCA que nous présentons est celui qui permet l'utilisation de Structures de Données Distribuées (*SDD*). Celles-ci profitent des multiples avantages offerts par l'utilisation des *spaces* dans les composants *ComputationSpace* et *WorkSpace* (cf. Figure 4.1).

4.4.4.1 Définition et utilisation de Structures de Données Distribuées

Parmi les différents types d'*entrée* disponibles dans un *ComputationSpace* (cf. Figure 4.17), le type *DDataStructure* permet de définir et d'utiliser des *SDD*. La classe éponyme fait partie d'un ensemble de classes abstraites et d'interfaces (cf. Figure 4.26) fournies par le framework *MCA*. Cette *API* spécifie la structure de classes pour la définition de nouveaux types de *SDD*. Celles-ci permet de partager des données structurées (matrices, vecteurs...) entre plusieurs agents *MCAWorker*. En réalité, ce ne sont pas les agents *MCAWorker* qui manipulent directement ces *SDD* mais les *ComputeAgent* exécutés par les agents *MCAWorker*. Pour simplifier la suite de la lecture, nous utilisons le terme *MCAWorker* pour désigner le couple *MCAWorker-ComputeAgent* qui s'exécute lors du traitement d'une tâche.

16. *JNI* (*J*ava *N*ative *I*nterface) [Lia99]. Celui-ci permet à du code Java d'appeler ou d'être appelé par des applications natives.

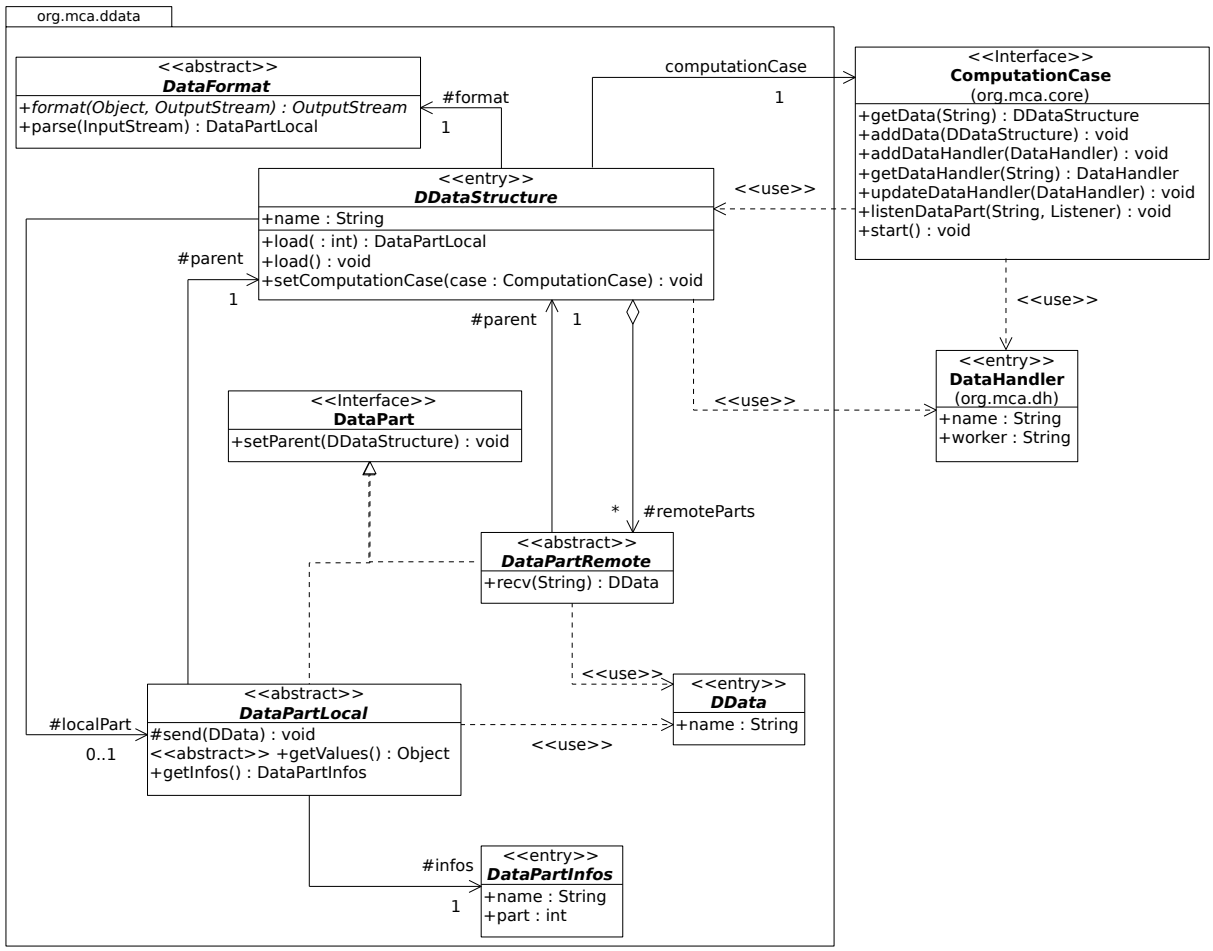


FIGURE 4.26 – Diagramme des classes du package `org.mca.ddata` définissant l'*API* dédiée à la définition de *SDD*.

Pour définir une *SDD* composée de n parties, il est nécessaire d'écrire $n + 1$ *entrées* dans un *ComputationSpace* :

– une *entrée* de type *DDataStructure*, comme par exemple avec le type `DMatrix` (cf. Section 4.4.4.2) ;

– n *entrées* de type *DataHandler*. Celles-ci permettent l'accès aux données de chaque partie de la *SDD*.

L'ensemble des *entrées* d'une même *SDD* respecte une convention de nommage. Par exemple, pour une *SDD* composée de n parties, si l'*entrée* de type *DDataStructure* a la valeur de sa propriété `name` égale à *dds*, alors la valeur de la propriété *name* de chacune des n *entrées* de type *DataHandler* est égale à $dds - i$, avec $i = 1, \ldots, n$.

La classe `DDataStructure` possède l'attribut `format` de type `DataFormat`. Cette attribut indique le format des données accessibles par les entrées de type *DataHandler*. Un instance de la classe `DataFormat` donne à chaque agent *MCAWorker* le moyen de lire, via la méthode `parse`, ou d'écrire, via la méthode `format`), les données d'une partie de la *SDD*. Par cette technique, un agent

MCAWorker peut lire et/ou écrire dans une *SDD* sans connaître à l'avance le format des données qu'elle contient.

Pour accéder aux données d'une *SDD*, un agent *MCAWorker* récupère l'*entrée* de type `DDataStructure` correspondante dans le *ComputationSpace*. Deux scénarios sont alors possibles, chacun correspondant à une méthode définie dans la classe `DDataStructure` :

– `public DataPartLocal load(int i)`
  Cette méthode permet de charger localement la partie i si elle n'a pas été déjà chargée par un autre agent *MCAWorker*. Elle est invoquée si la tâche récupérée par l'agent *MCAWorker* doit modifier les données de la *SDD*. En cas de succès, la méthode télécharge les données de la partie i dans le système de fichier local à la machine où s'exécute l'agent *MCAWorker*. En cas de succès, une *entrée* de type `DataPartInfos` est écrite dans le *WorksSpace* de l'agent *MCAWorker* (cette *entrée* est utile pour la gestion de la tolérance aux pannes dans les *SDD* (cf. Section 4.4.4.3) et retourne une instance de la classe `DataPartLocal`. Celle-ci représente un lien vers le *Workspace* de l'agent *MCAWorker*. La partie i est alors accessible en lecture et en écriture par cet agent *MCAWorker* et les autres parties de la *SDD* ne sont accessibles qu'en lecture via des instances de `DataPartRemote`. Ces instances sont des liens vers le *WorkSpace* de chaque agent *MCAWorker* ayant chargé une autre partie de la *SDD* et les données partagées sont lues via la méthode `recv` de la classe `DataPartRemote`. De la même façon, l'agent *MCAWorker* partage les données de la partie i avec les autres agents *MCAWorker*. Pour cela, il utilise son *WorkSpace* pour y déposer des entrées de type `DData` (cf. Figure 4.27) via la méthode `send` de la classe `DataPartLocal`.
– `public void load()`
  Cette méthode permet l'accès aux données de la *SDD* en lecture. Elle est invoquée si la tâche récupérée par l'agent *MCAWorker* ne doit pas modifier les données de la *SDD*. Dans ce cas, aucune partie n'est chargée localement par l'agent *MCAWorker* et toutes les données de la *SDD* sont accessibles en lecture via des instances de type `DataPartRemote`.

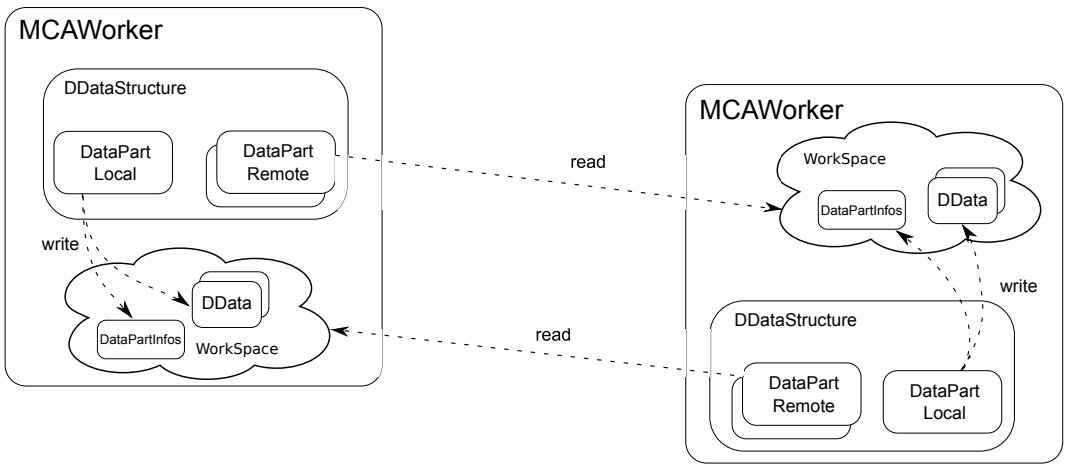


FIGURE 4.27 – Communication entre deux parties d'une Structure de Données Distribuée.

Notre framework *MCA* propose deux implantations de cette spécification sur les Structures de Données Distribuées : une pour la définition de matrices distribuées et une pour la définition de vecteurs distribués. Nous présentons dans la section suivante celle fournie pour utiliser des matrices distribuées.

4.4.4.2 Un exemple de *SDD* : une matrice distribuée

Le framework *MCA* propose l'implantation d'un type de *SDD* permettant de définir une matrice distribuée contenant des nombres réels de type `double`. L'implantation d'un type de *SDD* impose de définir une implantation pour chaque classe abstraite et interface de la figure 4.26. La figure 4.28 présente l'ensemble des classes permettant la définition et l'utilisation de matrices distribuées.

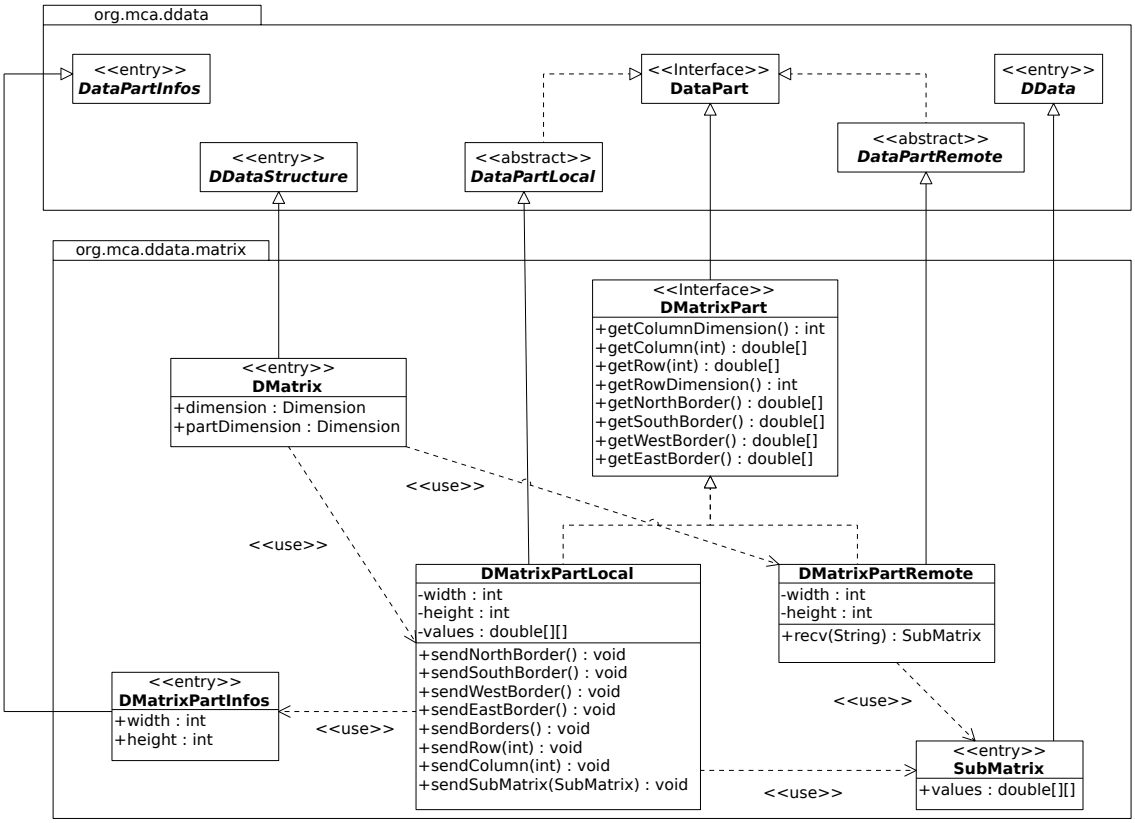


FIGURE 4.28 – Diagramme des classes pour la manipulation d'une matrice distribuée.

Dans le cas d'une matrice distribuée, la classe `DMatrix` définit un nouveau type d'*entrée* (c'est pourquoi ses attributs sont publics) qui représente ce type de *SDD*. Cette *entrée* permet de définir les dimensions [17] de la matrice entière (attribut `dimension`) et les dimensions de chaque partie (attribut `partDimension`). Chaque partie de ce type de *SDD* est accessible via une instances

17. La dimension d'une matrice se définit par le nombre de lignes et de colonnes de la matrice. La classe `Dimension` possède deux attributs : `width` pour le nombre de colonnes et `height` pour le nombre de lignes.

d'une classe implantant l'interface `DMatrixPart`. Cette interface définit des méthodes utiles pour récupérer des valeurs (colonnes, lignes, bordures, sous-matrices...) contenues dans la partie de la matrice correspondante . Deux classes implantent cette interface :

– la classe `DMatrixPartLocal` qui hérite de la classe `DataPartLocal`. Elle représente la partie locale à l'agent *MCAWorker*. Les valeurs de cette partie sont accessibles par l'attribut `values` et les implantations des méthodes définies dans l'interface `DMatrixPart` accèdent directement aux valeurs contenues dans cet attribut. La classe `DMatrixPartLocal` fournit également des méthodes pour partager les données (colonnes, lignes, bordures, sous-matrices...) locales avec les autres agents *MCAWorker*. Chacune de ces méthodes utilise des *entrées* de type `SubMatrix` qu'elles ajoutent au *WorkSpace* pour les mettre à disposition des autres agents *MCAWorker*.

– la classe `DMatrixPartRemote` qui hérite de la classe `DataPartRemote`. Elle représente une partie de la matrice prise en charge par un agent *MCAWorker* distant. Les implantations des méthodes définies dans l'interface `DMatrixPart` utilisent la méthode `recv` pour lire les *entrées* de type `SubMatrix` se trouvant dans le *WorkSpace* de l'agent *MCAWorker* responsable de cette partie distante.

La figure 4.29 modélise l'exemple d'une matrice distribuée sur 4 agents *MCAWorker*. Chaque agent *MCAWorker* a en charge une partie de cette matrice dont il sauvegarde les informations dans son *WorkSpace* à l'aide d'une *entrée* de type `DMatrixPartInfos`. Chaque partie est disponible via des instances de classes implantant l'interface `DMatrixPart`. Si la partie est locale à l'agent *MCAWorker* alors celui-ci utilise une instance de la classe `DMatrixPartLocal` pour accéder aux données de cette partie. Si la partie se trouve sur un autre agent *MCAWorker* alors elle est accessible par une instance de la classe `DMatrixPartRemote`. Chaque agent *MCAWorker* partage avec les autres agents *MCAWorker* les bords de sa partie locale via des *entrées* de type `SubMatrix`.

La section 5.2 utilise ce type de *SDD* et présente la configuration d'une matrice distribuée en vue de la résolution d'un cas de calcul suivant le modèle *SPMD*.

4.4.4.3 La tolérance aux pannes dans les structures de données distribuées

Comme tous les autres composants de l'architecture *MCA*, les *SDD* sont des structures tolérantes aux pannes. La panne ou l'indisponibilité d'un ou plusieurs agents *MCAWorker* en charge des parties d'une *SDD* ne doit pas rendre inaccessible les parties de la *SDD*. Il est important de garantir aux différents agents *MCAWorker* un accès permanent aux données d'une *SDD* partagée entre plusieurs agents *MCAWorker*.

La panne d'un agent *MCAWorker* entraine le passage de la partie de la *SDD* prise en charge par cet agent dans un état temporairement inaccessible. L'*entrée* de type *DataHandler* associée à cette partie est mise à jour (la valeur de son attribut `worker` devient nulle). Les autres agents *MCAWorker* n'ont plus accès à cette partie, elle devient alors libre pour qu'un autre agent *MCAWorker* puisse la prendre en charge. Deux scénarios sont alors possibles pour que les données de cette partie soient de nouveau accessibles :

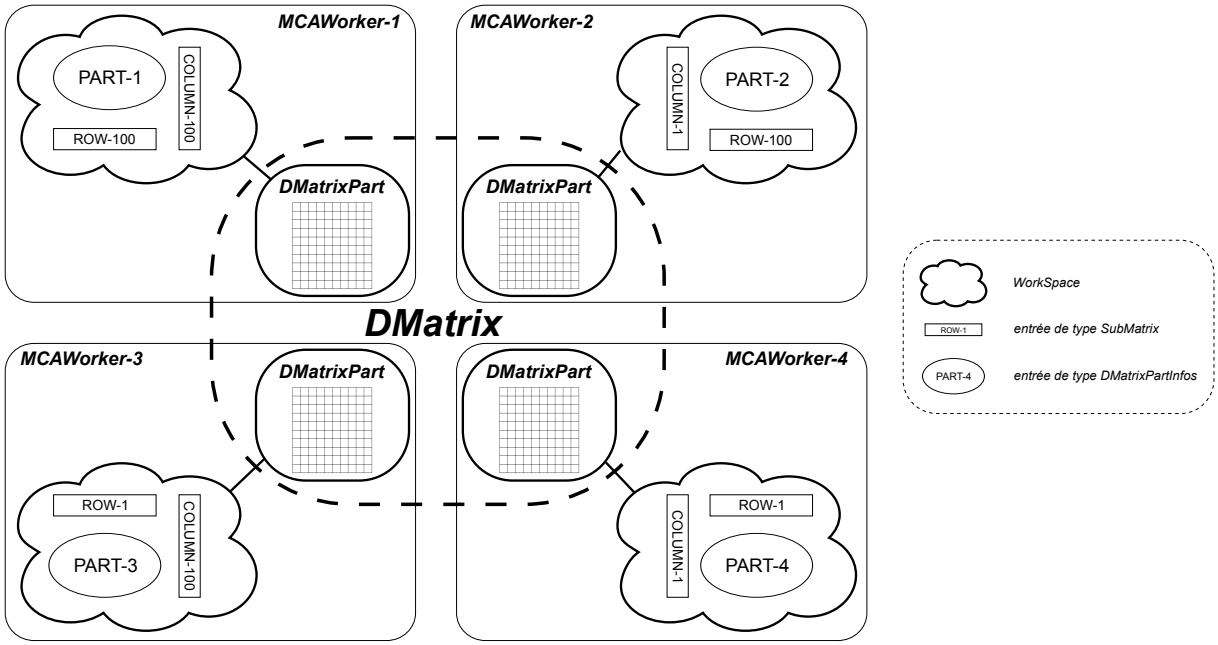


FIGURE 4.29 – Modélisation d'une matrice distribuée sur 4 agents *MCAWorker*.

– Le même agent *MCAWorker* redémarre avant qu'un autre agent *MCAWorker* ait pris en charge la partie devenue disponible. Grâce à l'entrée de type `DataPartInfos`, l'agent *MCAWorker* connaît la partie qu'il avait prise en charge avant son arrêt. Si la partie est disponible et n'a pas été modifiée depuis l'arrêt de l'agent, alors l'agent *MCAWorker* prend de nouveau en charge cette partie sans avoir à télécharger à nouveau les données.
– Un autre agent *MCAWorker* prend en charge la partie devenue libre. Ce cas est décrit dans la figure 4.30. Les instances de type `DataPartRemote` jouent un rôle important car elles surveillent les modifications apportées aux différents `DataHandler`, correspondant à chaque partie de la *SDD*, du *ComputationSpace*. Les agents *MCAWorker* qui ont un accès à cette partie peuvent à nouveau lire les données qu'elle contient.

4.5 Conclusion

Nous avons présenté dans ce chapitre le framework *MCA*. Celui-ci propose des outils pour la résolution de cas de calcul numérique dans un environnement distribué hétérogène. L'implantation de ce framework a été réalisée à partir de la modélisation formelle d'une architecture logicielle décrite dans le chapitre 2.

Notre architecture réalise nos deux objectifs clés : la transparence de panne et la transparence d'échelle (cf. Section 1.1.2). Effectivement, les composants *MCAWorker*, et leur *WorkSpace* associé, sont distribués de manière quelconque sur le réseau, leur nombre varie de manière dynamique sans nuire à la terminaison d'un cas de calcul. L'indisponibilité d'un agent *MCAWorker* est invisible à l'utilisateur. Les données d'un cas de calcul supportent deux autres transparences : la

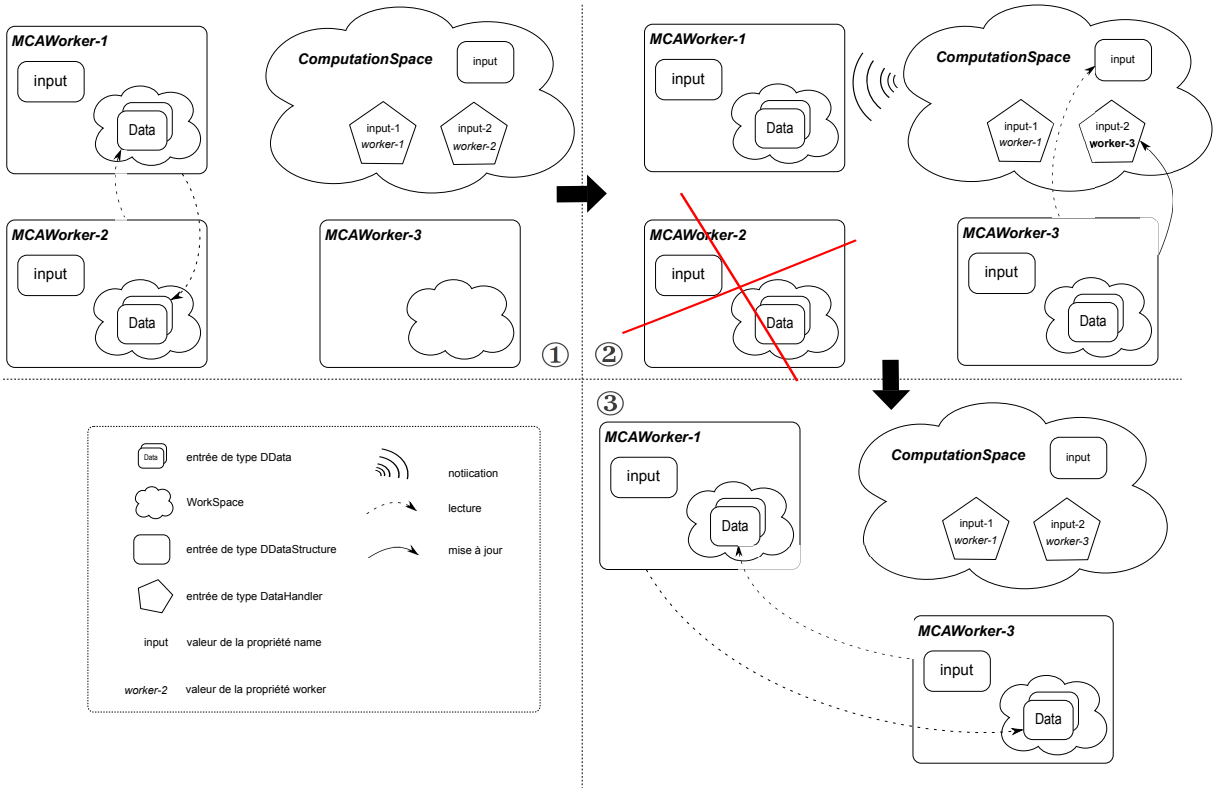


FIGURE 4.30 – Tolérance aux pannes dans les Structures de Données Distribuées. ① Partage d'une structure de données distribuées, nommée *input*, entre deux agents *MCAWorker* (*MCAWorker-1* et *MCAWorker-2*). ② Panne de l'agent *MCAWorker-2*, reprise de la partie disponible (*input-2*) par l'agent *MCAWorker-3* et notification des modifications à l'agent *MCAWorker-1*. ③ L'agent *MCAWorker-1* peut de nouveau lire les données contenues dans la partie *input-2*.

transparence d'accès, qui est mise en œuvre grâce à la notion de *DataPart* et de ses implantations locale et distante, et la transparence de localisation, mise en œuvre afin qu'un agent *MCAWorker* ne connaisse pas l'emplacement où les données sont conservées.

Pour la gestion des parties du code de calcul, nous avons mis en place une transparence de mobilité permettant au couple *MCAWorker-ComputeAgent* d'exécuter un code sans que la migration n'ait été visible des autres *MCAWorker*. Ainsi, la possibilité de déplacer du code natif est une avancée considérable par rapport aux outils de simulation numériques existants et apporte l'adaptation au contexte d'exécution.

Enfin, l'utilisation des *spaces* comme élément central de notre architecture supporte la transparence de réplication de données. Il apporte la persistance de ses données et l'utilisation de transactions distribuées.

Dans le chapitre suivant, nous évaluons le framework *MCA* en implantant différents cas de calcul et en les déployant sur la grille de calcul du LACL.

Chapitre 5

Mise en œuvre et expérimentations

Sommaire

Dans le chapitre 4, nous avons présenté notre framework nommé *MCA*. Celui-ci propose une *API* pour la résolution de cas de calcul numérique dans des environnements distribués hétérogènes comme, par exemple, les grilles de calcul (cf. Section 1.1). Dans ce chapitre, nous évaluons ce framework. Pour commencer, nous décrivons l'environnement d'exécution et la méthode d'évaluation du framework puis le déploiement d'un cas de calcul (Section 5.1). Les trois sections suivantes présentent chacune la résolution d'un cas de calcul pour mettre en évidence tour à tour une des propriétés de notre framework, dont certaines sont définies et vérifiées dans la chapitre 3. La résolution de l'équation de Laplace, réalisée dans la section 5.2, permet de démontrer la capacité d'un système basé sur le framework *MCA* à s'adapter à un environnement d'exécution

qui évolue au cours de la résolution d'un cas de calcul. Dans la section 5.3, nous présentons un cas de calcul de type *Maître-Travailleurs* avec l'approximation du nombre π par une méthode de *Monte Carlo* afin d'évaluer les performances offertes par le framework *MCA* en activant la tolérance aux pannes en présence ou non de pannes. Enfin, la section 5.4 met en avant le simplicité de développement offerte par l'*API MCA-Skel*, fournie par notre framework, dans le but de résoudre des cas de calcul utilisant des squelettes algorithmiques.

5.1 Mise en œuvre d'une plate-forme pour l'évaluation du framework MCA

5.1.1 Présentation de l'environnement d'expérimentation

Notre laboratoire, le *LACL*, dispose d'une grille de calcul disponible pour les expérimentations de ses membres. Nous utilisons cette grille pour mettre en place une plate-forme de calcul afin de tester notre framework *MCA*. Cette grille est constituée de 21 nœuds, reliés entre eux par un réseau Gigabit Ethernet. Elle est composée de :

– un nœud maître, nommé *bicephale*. Il est équipé d'un processeur double cœur Intel Pentium D cadencé à 3 Ghz et de 1 Go de mémoire vive. L'accès à la grille est réalisé à partir de ce nœud et aucun processus propre au framework *MCA* n'est exécuté dessus.

– 20 nœuds de calcul (nommés *node21* à *node40*), chacun équipé d'un processeur double cœur Intel Pentium E2180 cadencé à 2 GHz et de 2 Go de mémoire vive.

Le système d'exploitation installé sur chaque nœud est la distribution Linux Ubuntu 10.04 LTS.

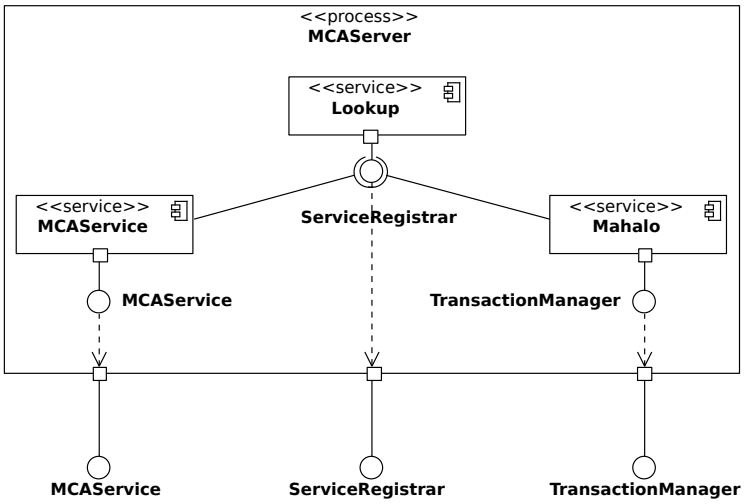


FIGURE 5.1 – Diagramme de composants du processus *MCAServer*.

Pour les expérimentations présentées dans ce chapitre, nous déployons deux types de composants issus de notre framework sur la grille :

– le composant *MCAServer* (cf. Figure 5.1) sur 3 nœuds (*node21* à *node23*), nous utilisons ici la redondance spatiale (cf. Section 1.3.3) pour rendre ce composant tolérant aux pannes. Ce composant est constitué des services *Lookup* (cf. Section 4.2.1.1) et *Mahalo* (cf. Section 4.3.3.1) fournis par Jini™ et du service *MCAService* (cf. Section 4.4.1.1). Les services *Mahalo* et *MCAService* enregistre chacun un proxy dans l'annuaire proposé par le service *Lookup*. Ces trois services sont exposés sur le réseau et sont accessibles par tous les nœuds de la grille.
– le composant *MCAWorker* (cf. Section 4.4.2) sur 17 nœuds (*node24* à *node40*).

La figure 5.2 présente le diagramme de déploiement de ces deux types de composant sur la grille du LACL. Nous remarquons sur ce diagramme que chacun de ces deux composants est implémenté par un artéfact : *mca-server.jar* pour le composant *MCAServer* et *mca-worker.jar* pour le composant *MCAWorker*. La relation entre un composant et l'artéfact qui l'implémente est appelé *manifestation* et est modélisé sur la figure 5.2 par une relation de dépendance stéréotypée *« manifest »* .

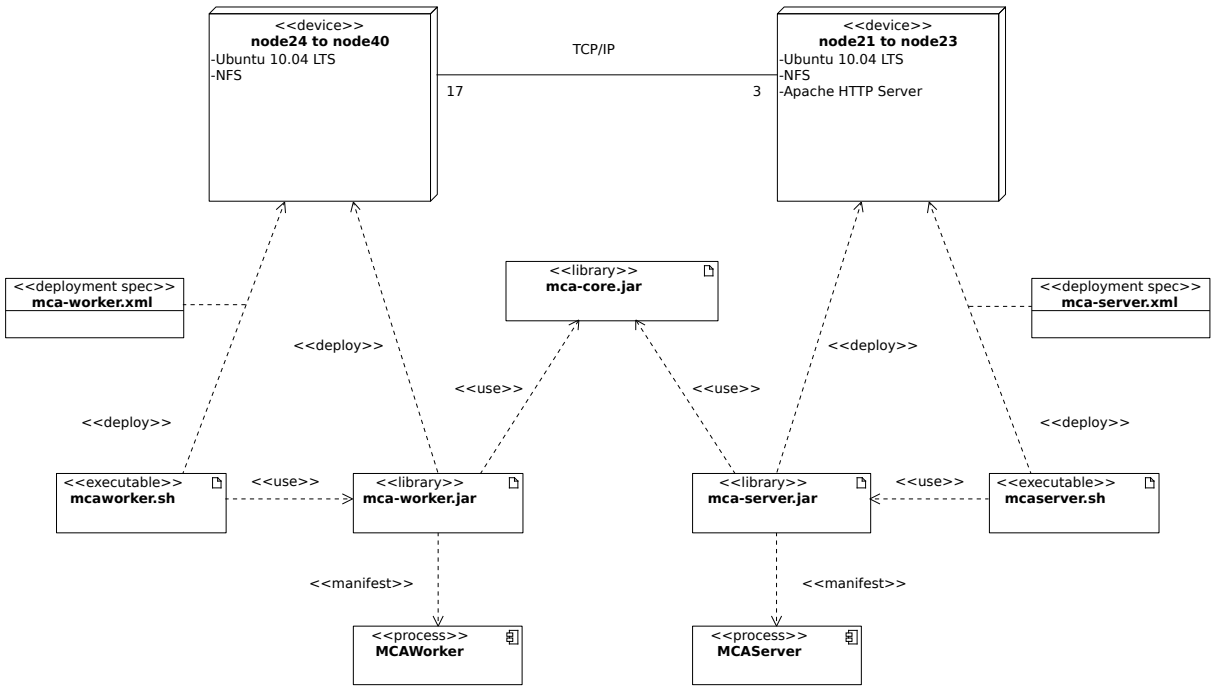


FIGURE 5.2 – Diagramme de déploiement des composants *MCAWorker* et *MCAServer* sur la grille de calcul du LACL.

En plus des librairies Java qui implémentent chaque composant à déployer, nous déployons sur le même nœud un script d'exécution (*mcaworker.sh* ou *mcaserver.sh*) et un fichier de configuration (*mca-worker.xml* ou *mca-server.xml*) spécifique au composant. Le fichier de configuration est au format *XML* et configure l'instance du composant déployé sur le nœud (services à démarrer, répertoires utilisés...).

Notons la présence d'un serveur WEB (*Apache Http Server* sur les nœuds où est déployé le composant *MCAServer* (cf. Figure 5.2)). Ces instances permettent aux agents *MCAWorker* de

télécharger les libraires nécessaires à l'exécution des agents mobiles de type *ComputeAgent* (cf. Section 4.4.3). En effet, l'environnement d'exécution des agents *MCAWorker* contient uniquement les librairies du framework *MCA* définies par les fichiers *mca-core.jar*, pour le code commun avec le composant *MCAServer* et *mca-worker.jar*, pour le code propre au composant *MCAWorker*. Pour chaque cas de calcul que nous présentons dans la suite de ce chapitre, nous déposons les libraires Java associées aux agents de type *ComputeAgent* dans le répertoire publié par le serveur WEB sur les nœuds *node21*, *node22* et *node23*.

L'administration de notre plate-forme de calcul est réalisée à partir du nœud *bicephale*. Celle-ci est simplifiée grâce au service *NFS* (pour *Network File System*) démarré sur chaque nœud de la grille. En effet, le répertoire d'un compte utilisateur créé sur *bicephale* est alors répliqué sur tous les nœuds de la grille. Nous avons créé l'utilisateur *mca* et placé tous les fichiers (librairies, scripts d'exécution, fichier de configuration...) utiles à l'exécution de notre plate-forme dans le répertoire de cet utilisateur (`/home/mca/`). Ainsi, la modification d'un fichier de configuration (comme par exemple *mca-worker.xml* ou *mca-server.xml*) à partir de *bicephale* impacte tous les composants déployés sur le nœud de la grille.

5.1.2 Méthodologie d'évaluation du framework

Pour chaque expérimentation que nous proposons dans la suite de ce chapitre, nous effectuons des mesures de temps d'exécution lors de la résolution des cas de calcul. Pour réaliser ces mesures, nous avons choisi d'utiliser la librairie *JETM* (pour *Java Execution Time Measure*) [JET]. Celle-ci propose une *API* fournissant un ensemble de classes qui permettent d'instancier des objets chargés de collecter et d'agréger des informations contenues dans des points de mesure. L'utilisation de cette *API* est présentée par la figure 5.3. Pour démarrer la collecte de points de mesure, il est nécessaire de récupérer une instance de type `EtmMonitor` et d'invoquer la méthode `start` sur cette instance. La mesure d'un temps d'exécution est alors réalisée à l'aide d'une instance de la classe `EtmPoint`. L'invocation de la méthode `collect` collecte le temps d'exécution depuis la création du point de mesure.

Pour réaliser des mesures de temps d'exécution lors de la résolution de cas de calcul, nous devons donc ajouter des points de mesure dans le code défini par les *ComputeAgent*. Nous souhaitons réaliser ces mesures sans modifier le code source de ces agents mais fournir à un *ComputeAgent* les informations nécessaires pour que l'agent *MCAWorker* qui l'exécute puisse faire des mesures de temps spécifiques à ce *ComputeAgent*, comme par exemple le temps d'exécution de certaines méthodes. Les mécanismes proposés par la programmation orientée aspects et l'*API* d'instrumentation offerte par Java nous permettent de répondre à nos exigences.

5.1.2.1 La programmation orientée aspect

La *Programmation Orientée Aspect* (*POA*) [PRS04] est un paradigme de programmation qui permet de séparer les différentes préoccupations techniques lors du développement d'une appli-

```
1  // Création et démarrage du collecteur de points de mesure
2  EtmMonitor etmMonitor = EtmManager.getEtmMonitor();
3  etmMonitor.start();
4  // Création d'un point de mesure nommé ptM1
5  EtmPoint point = etmMonitor.createPoint("ptM1");
6  // Collecte du point de mesure
7  point.collect();
8  // Visualisation des résultats
9  etmMonitor.render(new SimpleTextRenderer());
10 // Arrêt du collecteur
11 etmMonitor.stop();
```

FIGURE 5.3 – Exemple d'utilisation de le librairie *JETM*

cation. Via les mécanismes offerts par ce type de développement, nous allons pouvoir séparer le code propre à la résolution d'un cas de calcul (code dit « métier ») du code utile à la mesure des temps d'exécution (aspect technique). La *POA* définit un ensemble de *points de jonction (joinpoint)* pour établir un lien entre le code dit métier et les différents aspects. Ces points de jonction définissent les différentes parties d'un programme où il est possible d'insérer du code, comme par exemple l'appel ou l'exécution d'une méthode, la lecture ou la modification d'un attribut, la levée ou la capture d'une exception. Un *greffon (advice)* est le code qui peut être inséré avant, après ou autour de ces différents points de jonction. Pour lier différents points de jonction à un greffon, il existe la notion de *coupe (pointcut)*. Une coupe définit l'ensemble des points de jonction où est inséré le code contenu dans le greffon. Un *aspect* est le composant logiciel qui définit un ensemble de greffons et y associe des coupes.

Une fois l'aspect défini, il est important de choisir une technique de tissage. Le processus de *tissage* (*weaving*) est l'action qui insère le code des greffons dans le code métier aux différents points de jonction définis par la coupe. Il existe différentes techniques de tissage :

- *Compile Time Weaving* : Le tissage se fait lors de la compilation. Il est alors nécessaire d'utiliser un compilateur différent du compilateur standard.
- *Bytecode Time Weaving* : Le tissage se fait sur les classes déjà compilées. Ici, il n'est pas utile de recompiler le code source si l'on modifie les aspects à activer.
- *Load Time Weaving (LTW)* : Le tissage se fait lors du chargement des classes pendant l'exécution du code. C'est celui que nous utilisons pour faire nos mesures.

Si de nombreux tisseurs d'aspects sont disponibles, comme *JAC* et *Spring AOP*, notre choix s'est porté vers l'outil *AspectJ*, proposé par le projet *Eclipse*, car il est celui qui répond le plus à nos besoins. En particulier, dans le fait de proposer un tissage de type *LTW* adapté pour l'utilisation d'aspect dans du code mobile. Celui-ci utilise l'*API* d'instrumentation fournie par le *JDK*.

5.1.2.2 Instrumentation des agents *MCAWorker*

Introduits avec la version 5 du *JDK*, les « agents » Java sont des programmes Java lancés au démarrage de la JVM [18]. Lors de l'exécution d'une application Java, ce type d'agent permet de modifier le bytecode d'une classe en interceptant son chargement réalisé par la *JVM*. C'est par ce mécanisme que *AspectJ* réalise du *Load Time Weaving* en fournissant un agent de ce type. Cet agent permet de modifier le bytecode des classes en injectant les aspects déclarés dans un fichier nommé `aop-ajc.xml`. Celui-ci doit se situer dans le répertoire META-INF d'un des fichiers jar chargés par la *JVM*.

Nous avons développé l'aspect `ExecutionTimeAspect` (cf. Figure 5.4) pour permettre la mesure de temps d'exécution de méthodes. Cet aspect définit deux greffons via les méthodes `monitoring` et `collectMeasurePoint`. Ceux-ci sont déclarés avec l'annotation `@Around`, ce qui signifie qu'il est possible d'ajouter le code du greffon avant et après un point de jointure défini par la coupe associée au greffon. Ces deux greffons utilisent la librairie *JETM* comme ceci :

- Le greffon `monitoring` démarre la collecte de points de mesure avant le point de jonction, affiche les mesures puis stoppe la collecte après le point de jonction.
- Le greffon `collectMeasurePoint` crée un point de mesure avant le point de jonction et collecte le temps d'exécution après le point de jonction.

L'aspect `ExecutionTimeAspect` définit également deux coupes (via l'annotation `@Pointcut`) :

- la coupe `monitoringMethod` associée au greffon `monitoring`,
- la coupe `measuringMethod` associée au greffon `collectMeasurePoint`.

AspectJ permet de définir des aspects abstraits. Nous utilisons cette technique pour définir l'aspect `ExecutionTimeAspect` abstrait. En effet, l'aspect `ExecutionTimeAspect` ne peut être utilisé tel qu'il est défini dans la figure 5.4 car les coupes déclarées ne définissent aucun point de jonction. Pour utiliser les greffons définis dans cet aspect abstrait, il est nécessaire de définir un aspect dit « concret » dans un fichier `aop-ajc.xml` utilisé par l'agent java fourni par *AspectJ*. La figure 5.5 présente un exemple de fichier `aop-ajc.xml` qui définit un aspect héritant de `ExecutionTimeAspect` pour ajouter deux points de jonction à la coupe `measuringMethod`.

La technique que nous venons de présenter est celle utilisée pour évaluer notre framework MCA. Notons que les points de mesure varient selon les cas de calcul, c'est donc aux agents mobiles de type *ComputeAgent* de définir les points de jonction de la coupe `measuringMethod`. Pour cela, le fichier *jar* contenant les classes d'un *ComputeAgent* contient également un fichier `aop-ajc.xml`. Ainsi, après avoir téléchargé le jar contenant les classes utiles à l'exécution d'un *ComputeAgent*, la *JVM* dans laquelle s'exécute l'agent *MCAWorker* réalise le tissage à l'aide de l'agent Java fourni par *AspectJ*. La figure 5.6 présente les différents composants utilisés pour réaliser ce tissage.

Pour réaliser des mesures dans les différents cas de calcul que nous présentons dans la suite de ce chapitre, nous ajoutons un fichier `aop-ajc.xml` aux fichiers *jar* contenant les classes des *Com-*

18. en ajoutant l'option *-javaagent :...* à la commande *java*

```
1  package org.mca.aspect;
2
3  public @Aspect abstract class ExecutionTimeAspect {
4
5   public @Pointcut void measuringMethod(){}
6   public @Pointcut void monitoringMethod() {}
7
8     private static final EtmMonitor etmMonitor = EtmManager.getEtmMonitor();
9
10  @Around(value="measuringMethod()")
11  public Object collectMeasurePoint(ProceedingJoinPoint joinPoint) throws Throwable {
12   String className = joinPoint.getSignature().getDeclaringType().getName();
13   String methodName = joinPoint.getSignature().getName();
14   EtmPoint point = etmMonitor.createPoint(className + ":" + methodName);
15   Object[] args = joinPoint.getArgs();
16   Object o = joinPoint.proceed(args);
17   point.collect();
18   return o;
19  }
20
21  @Around(value="monitoringMethod()")
22  public Object monitoring(ProceedingJoinPoint joinPoint) throws Throwable{
23   Aggregator aggregator = new MCAAggregator(new RootAggregator());
24   BasicEtmConfigurator.configure(true, aggregator);
25   etmMonitor.start();
26   Object[] args = joinPoint.getArgs();
27   Object o = joinPoint.proceed(args);
28   etmMonitor.render(new MCARenderer());
29   etmMonitor.stop();
30   return o;
31  }
32 }
```

FIGURE 5.4 – Définition de l'aspect abstrait `ExecutionTimeAspect`.

puteAgent exécutés lors du traitement des tâches de calcul. Ce fichier est configuré en fonction des méthodes à observer pour fournir des mesures de performances désirées.

5.1.3 Description et déploiement d'un cas de calcul

Chaque cas de calcul présenté dans les sections 5.2, 5.3 et 5.4 est déployé sur l'environnement d'expérimentation décrit dans la section 5.1.1. Le déploiement de chaque cas de calcul est configuré dans fichier XML appelé descripteur de déploiement (cf. Section 5.1.3.1). La commande `mca-deploy`, fournie par notre framework *MCA*, analyse ce fichier et déploie le cas de calcul sur la plate-forme de calcul *MCA*.

```
<!DOCTYPE aspectj PUBLIC
    "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
  <aspects>
    <concrete-aspect  name="org.mca.example.jacobi.aspect.JacobiExecutionTimeAspect"
          extends="org.mca.aspect.ExecutionTimeAspect">
      <pointcut name="measuringMethod"
    expression="execution(* org.mca.example.jacobi.agent.JacobiAgent.program(..)) ||
          execution(* org.mca.example.jacobi.agent.JacobiAgent.internalCompute(..))" />
    </concrete-aspect>
  </aspects>
</aspectj>
```

FIGURE 5.5 – Exemple d'un fichier `aop-ajc.xml` avec la définition d'un aspect héritant de l'aspect abstrait `ExecutionTimeAspect` (cf. Figure 5.4).

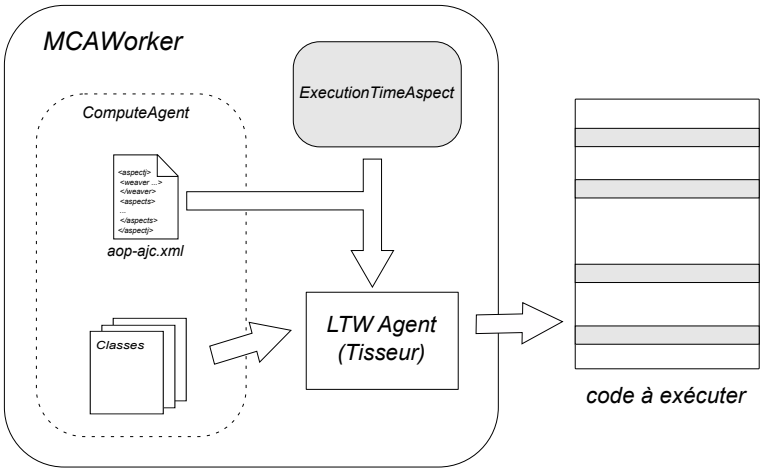


FIGURE 5.6 – Tissage d'aspect pour la mesure de temps d'exécution.

5.1.3.1 Le descripteur de déploiement

Le déploiement d'un cas de calcul est décrit dans un fichier de configuration au format XML. Ce fichier est appelé le *descripteur de déploiement* et sa structure est définie par un schéma XML modélisé par le diagramme de la figure 5.7. Nous pouvons noter qu'il est possible de définir les information suivantes :

- un nom (attribut `name` de la balise `case`) et une description du cas de calcul (balise `description`) ;
- des propriétés globales au cas de calcul (balise `properties`) ;
- des agents de type *ComputeAgent* (balise `<agents />`). Nous détaillons la définition d'un *ComputeAgent* dans la suite de cette section ;
- des structures de données distribuées (balise `<data />`) (cf. Section 4.4.4). Nous donnons un exemple de configuration dans la section 5.2 avec l'utilisation d'une matrice distribuée ;

– un type de déploiement soit avec la balise `<deployer />` et l'attribut `class` associé s'il s'agit d'un déploiement spécifique ou avec les balises `<spmd-deployer />`, `<mw-deployer />` ou `<skel-deployer />` si le cas de calcul respecte un modèle connu. Nous détaillons les différents types de déploiement dans la section 5.1.3.2.

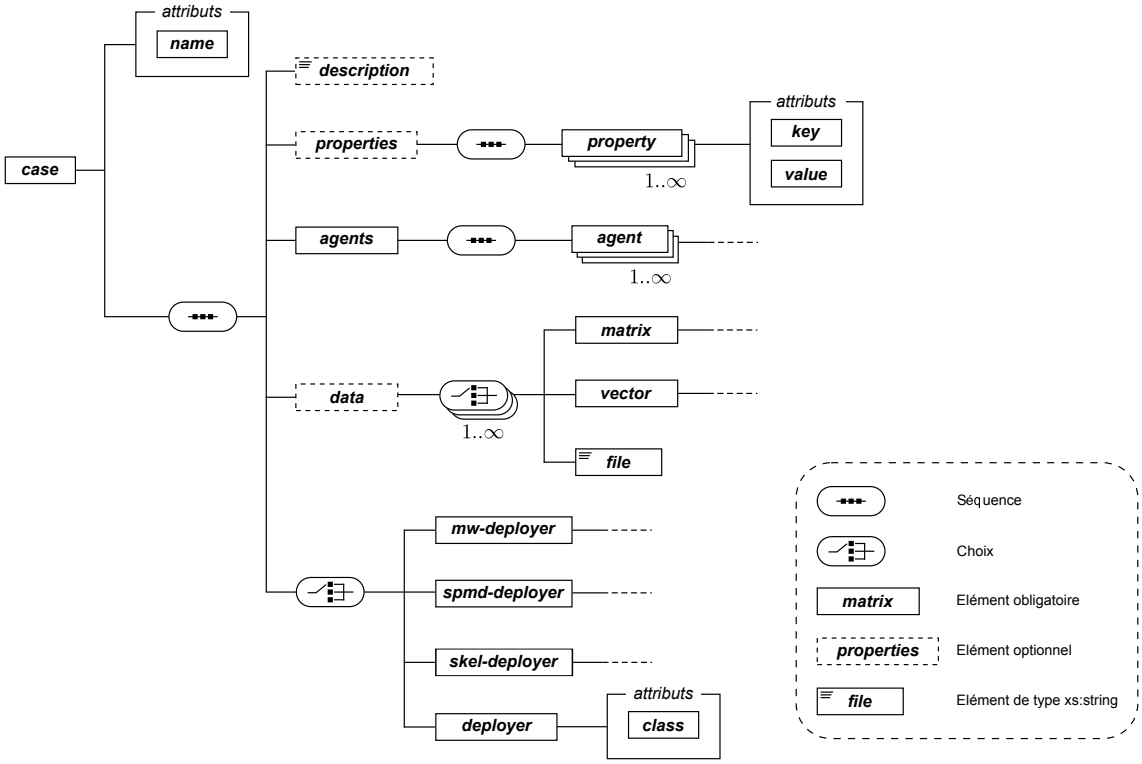


FIGURE 5.7 – Modélisation du schéma XML d'un descripteur de déploiement.

La figure 5.8 présente le contenu XML d'un descripteur de déploiement qui définit un cas de calcul nommé `c1` possédant une propriété globale `p1` (ce type de propriété est spécifié par le terme *Property* (eq. 2.48 dans le chapitre 2). La configuration déclare que le déploiement du cas de calcul est réalisé par une instance de la classe `ExampleCaseDeployer`, classe fille de la classe `ComputationCaseDeployer` (cf. Figure 5.10).

Chaque agent de type *ComputeAgent* est déclaré via le descripteur de déploiement à l'aide d'une balise `<agent />`. Le type XML qui définit cette balise est modélisé par le diagramme de la figure 5.9. A partir de ce diagramme, nous pouvons lister les différents éléments à fournir pour configurer un *ComputeAgent* :

– un *nom* (attribut `name` de la balise `<agent />`). Ce nom facilite la recherche de l'agent sur le réseau par les agents *MCAWorker*.
– un nom de *classe* (attribut `class` de la balise `<agent />`). Cette classe doit hériter, directement ou non, de la classe `AbstractComputeAgent` (cf. Figure 4.23).
– un *classpath* (balise `<classpath />`) : liste des chemins vers les répertoires et/ou les fichiers *jar* contenant les classes et interfaces utiles au déploiement de l'agent.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <case name="c1"
3    xmlns="http://www.lacl.fr/schema/mca"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://www.lacl.fr/schema/mca http://www.lacl.fr/schema/mca-1.0.xsd" >
6    <description>Exemple de CC</description>
7      <properties>
8        <property key="p1" value="v1"/>
9      </properties>
10     <agents>
11       <!-- Déclaration des différents ComputeAgent -->
12     </agents>
13     <deployer class="org.mca.example.ExampleCaseDeployer"/>
14 </case>
```

FIGURE 5.8 – Exemple d'un descripteur de déploiement.

- un *codebase* (balise `<codebase />`) : liste des URLs indiquant l'emplacement de fichiers *.class* ou de fichiers *jar* contenant les classes et interfaces utiles à l'exécution de l'agent par un agent *MCAWorker*.
- un *locator* (balise `<locator />`) : il s'agit du nom d'hôte (et éventuellement du numéro d'un port d'écoute) où se situe un annuaire *Jini* (via le service *Lookup*). L'agent est enregistré dans cet annuaire comme décrit dans la section 4.2.2.
- un fichier (balise `<bytecode />`) contenant du *bitcode* LLVM (cf. Section 4.4.3.2). Cette information est prise en compte uniquement si l'agent est défini comme un *ComputeAgent* natif via l'attribut `name` de la balise `<agent />`.

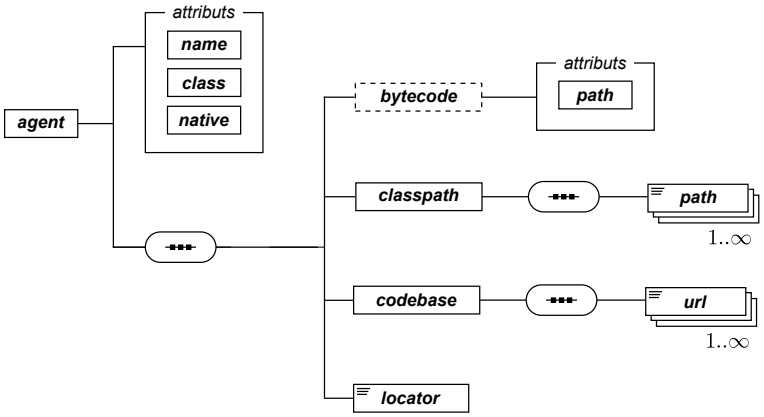


FIGURE 5.9 – Modélisation du schéma XML du type définissant un *ComputeAgent*.

5.1.3.2 Le déploiement d'un cas de calcul

L'interprétation du descripteur de déploiement est réalisée via le script `mca-deploy`. L'exécution de ce script donne lieu à la création d'une instance d'une classe héritant de la classe abstraite `ComputationCaseDeployer` (cf. Figure 5.10). Cette classe définit la méthode `deploy` qui réalise les étapes nécessaires au déploiement d'un cas de calcul sur la plate-forme de calcul *MCA*. Cette classe est abstraite, ce qui impose au développeur d'implanter la méthode `deployTasks`. Par cette méthode, le développeur doit ajouter les *entrées* de type *Task* au *ComputationSpace* associé au cas de calcul (cf. Section 4.4.1.2). Notons, dans le diagramme de la figure 5.10, la présence de trois implantations de la classe `ComputationCaseDeployer` : les classes `SPMDCaseDeployer`, `MWCaseDeployer` et `SkeletonCaseDeployer`. Ces classes sont fournies aux développeurs par le framework *MCA* et facilitent le déploiement des cas de calcul qui suivent respectivement les modèles *SPMD* (cf. Section 5.2), *Maître-Travailleurs* (cf. Section 5.3) ou à base de squelettes algorithmiques (cf. Section 5.4.1). Il est aussi possible de définir une autre implantation de la classe `ComputationCaseDeployer` si le cas de calcul ne respecte pas un des trois modèles déjà implantés. Une telle situation peut se produire avec un cas de calcul devant s'effectuer sur une architecture imposée, par exemple avec un découpage en blocs de données spécifiques.

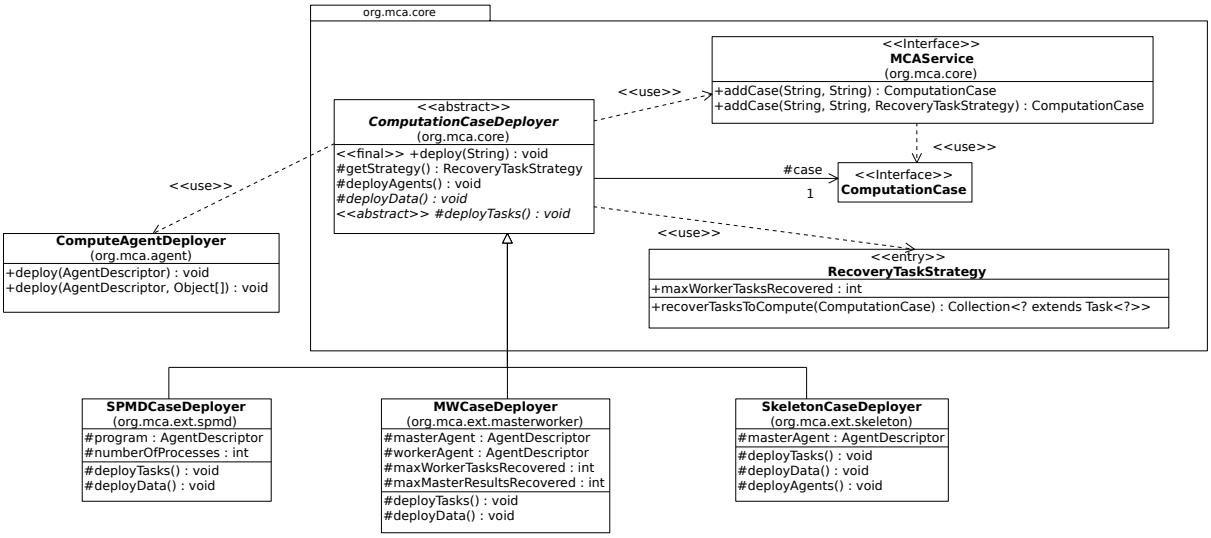


FIGURE 5.10 – Diagramme des classes utilisées lors du déploiement d'un cas de calcul sur la plate-forme *MCA*.

À la lecture du diagramme de séquence présenté dans la figure 5.11, nous pouvons définir les différentes étapes lors du déploiement d'un cas de calcul :

1. Recherche d'une instance du service *MCAService* disponible sur le réseau (cf. Section 4.4.1.1)
2. Appel de la méthode `addCase` sur l'instance de *MCAService*
3. Ajout des propriétés globales du cas de calcul en utilisant l'agent *ComputationCase* associé à ce nouveau cas de calcul

4. Enregistrement des agents de type *ComputeAgent*, déclarés dans la balise `<mca:agents />`, sur le réseau
5. Ajout des données déclarées dans la balise `<mca:data />`
6. Ajout des tâches du cas de calcul à l'aide de la méthode `deployTasks`.

Une fois la méthode `start` exécutée, le cas de calcul passe à l'état ACTIVE. Les agents *MCA-Worker* disponibles sont alors avertis grâce à leur moniteur d'activité (cf. Section 4.4.2).

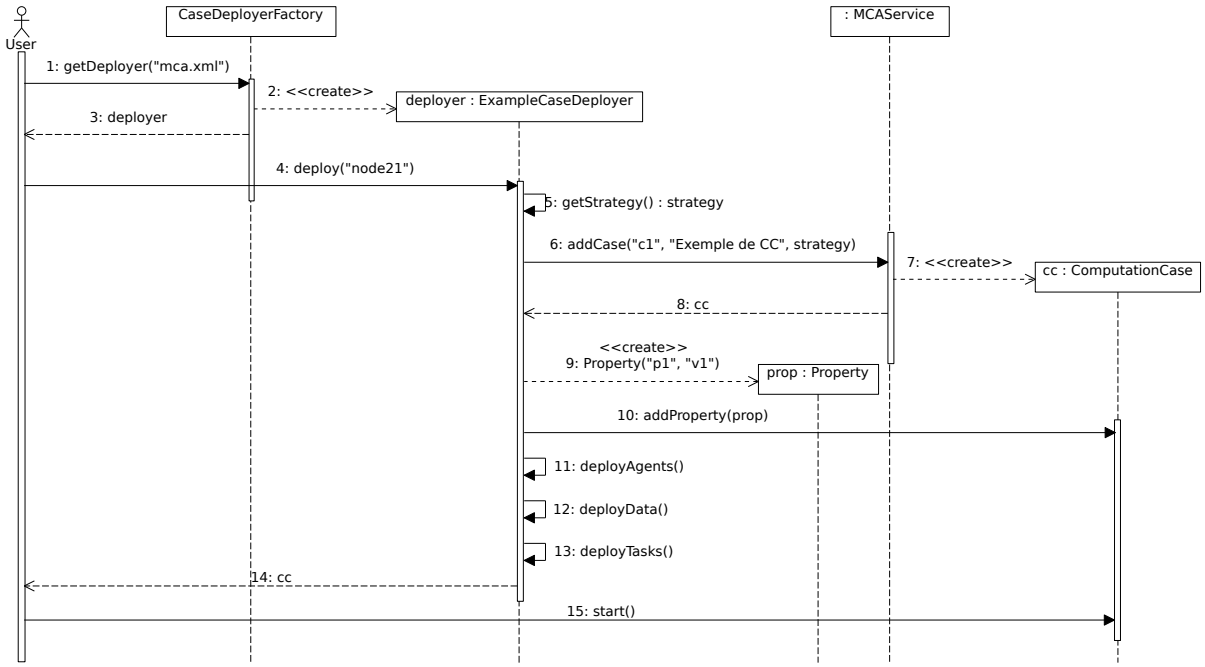


FIGURE 5.11 – Diagramme de séquence du déploiement du cas de calcul défini par le descripteur de déploiement de la figure 5.8 sur la plate-forme MCA installée sur la grille du LACL.

5.2 Évaluation de l'adaptabilité fournie par le framework *MCA*

L'objectif de cette section est d'évaluer la capacité d'un système basé sur le framework *MCA* à s'adapter à un environnement d'exécution qui évolue au cours de la résolution d'un cas de calcul. Nous présentons dans cette section l'*API MCA-SPMD* proposée par notre framework *MCA* qui permet la mise en place de cas de calcul suivant le modèle *SPMD* (cf. Section 1.1.3.1). Dans la section 5.2.1, nous illustrons cette *API* en implantant l'algorithme, appelé *itération de Jacobi*, pour résoudre l'équation de Laplace. Ce cas de calcul nous permet ensuite d'évaluer, dans la section 5.2.3, le surcoût induit par l'utilisation du framework *MCA*.

5.2.1 Présentation de l'*API MCA-SPMD*

Nous avons développé une *API* pour faciliter la mise en place d'une résolution de cas de calcul suivant le modèle *SPMD* (cf. Section 1.1.3.1). Les différents éléments de cette *API* répondent

aux besoins demandés par le modèle *SPMD* qui sont :

- Identification de chaque processus, représenté par un couple *ComputeAgent-Task* et exécuté par un agent *MCAWorker* (cf. Section 5.2.1.1), qui prend part au calcul avec la notion de *rang*. Cet identifiant donne la position du processus par rapport aux autres. La disposition des processus peut être basique ou suivre une organisation plus complexe avec la définition de topologies (cf. Section 5.2.1.2).
- Définition du programme unique exécuté par chaque processus en fonction du rang du processus qui exécute le programme, représenté par un *ComputeAgent*.
- Utilisation d'un ensemble d'opérations collectives pour permettre la communication et la synchronisation (cf. Section 5.2.1.3) entre les participants.

5.2.1.1 Un *ComputeAgent* adapté au modèle *SPMD*

L'*API MCA-SPMD* propose un ensemble de classes qui aide le développeur à mettre en place un cas de calcul suivant le modèle *SPMD*. L'association d'un *ComputeAgent*, spécialisé par une classe héritant de la classe abstraite `SPMDAgent` (cf. Figure 5.12), et d'une *entrée* de type *Task*, spécialisée par la classe `SPMDTask`, représente un processus participant à un calcul de type *SPMD*. La classe `SPMDTask` apporte la notion de *rang* à une *entrée* de type *Task*. Cette information est utilisée par la classe `SPMDAgent`, et toutes les classes qui en héritent, en particulier dans l'implantation de la méthode `program`. Cette méthode contient le code unique exécuté par chaque processus comme le propose le modèle *SPMD*.

Le déploiement d'un cas de calcul suivant le modèle *SPMD* est réalisé par une instance de la classe `SPMDCaseDeployer` (cf. Figure 5.10). Il consiste à déployer un unique *ComputeAgent*, de type `SPMDAgent` et à ajouter autant d'*entrée* de type `SPMDTask` qu'il y a de processus, au sens *SPMD* et non pas des *MCAWorker*, nécessaires à la résolution du cas de calcul. Le nombre de *MCAWorker* reste indépendant du nombre de tâches définies pour le cas de calcul.

5.2.1.2 Topologies cartésiennes

Les topologies cartésiennes fournissent un moyen d'organiser les différents processus qui participent à la résolution d'un cas de calcul suivant le modèle *SPMD*. Ces topologies facilitent les communications entre les processus (modélisés par les couples `SPMDTask-SPMDAgent`) exécutés par les agents *MCAWorker*. Lors de la définition d'un *ComputeAgent* de type `SPMDAgent`, le développeur doit implanter la méthode `getTopology` pour définir la topologie sur laquelle sont disposés les différents processus du cas de calcul. Cette organisation permet à chaque processus d'accéder de façon simple à ses processus voisins, le processus courant étant représenté par le couple `SPMDTask-SPMDAgent` exécuté par le *MCAWorker*) et de partager des données avec eux. La classe `SPMDProcess` modélise un processus voisin et permet de récupérer des données partagées par ce processus à l'aide de la méthode `recv`. Cette méthode se connecte au *WorkSpace* (cf. Section 4.4.2) de l'agent

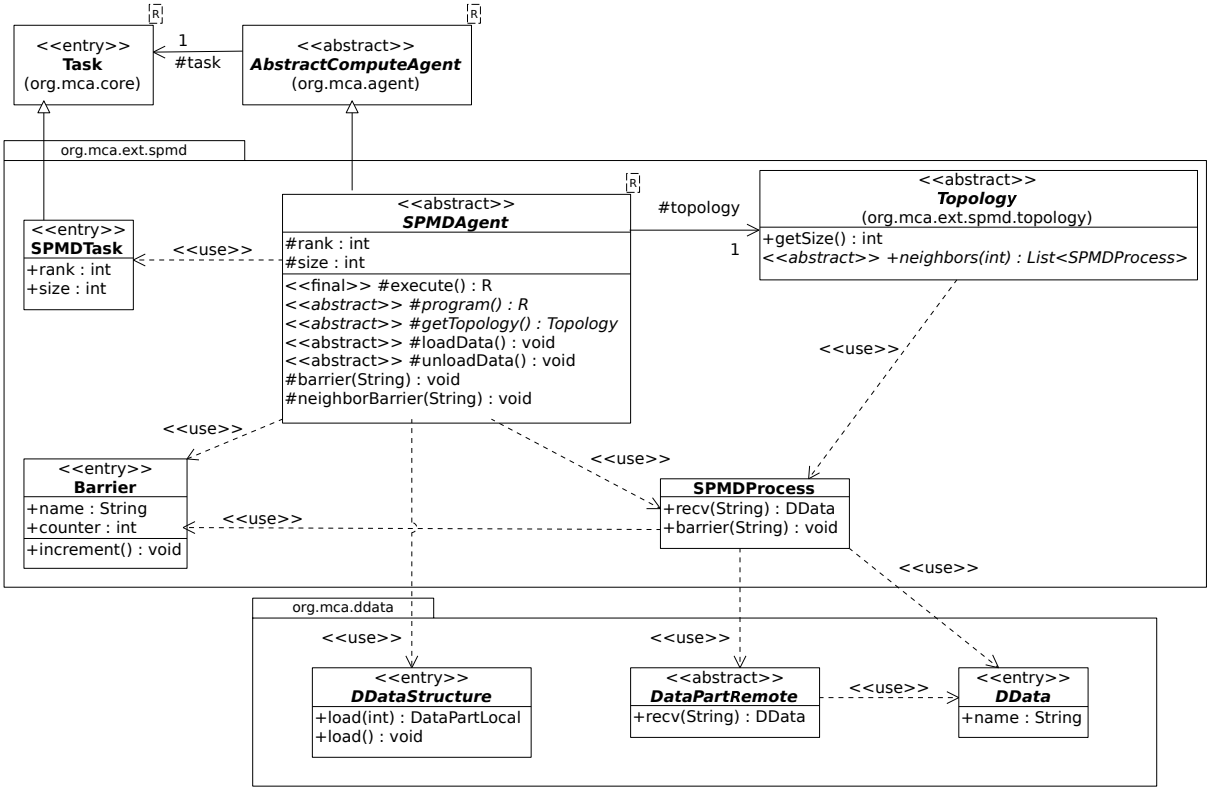


FIGURE 5.12 – Diagramme de classe de l'*API MCA-SPMD*

MCAWorker exécutant ce processus voisin et lit l'*entrée* de type *DData* correspondant au nom passé en paramètre.

La classe abstraite `Topology` (cf. Figure 5.13) est la classe de base pour définir une topologie. L'*API MCA-SPMD* propose différents types de topologie au développeur. Celui-ci peut les utiliser pour organiser les processus au sein d'un cas de calcul *SPMD*. Chaque type de topologie est représenté par une classe qui hérite de la classe `Topology`. Toutes les méthodes définies dans les sous-classes de `Topology` ont un paramètre de type `int`, noté `rank`. Celui-ci est égal au rang du processus courant traité par l'agent *MCAWorker*. Cette information permet de situer le processus dans la topologie utilisée pour la cas de calcul auquel il participe. Ce processus est alors le processus *actif* de l'agent *MCAWorker* qui exécute le *ComputeAgent*. La topologie utilisée pour organiser les processus d'un cas de calcul *SPMD* est définie par l'instance de type `Topology` renvoyée par l'implantation de la méthode `getTopology` lors de la définition d'un type *ComputeAgent* héritant de `SPMDAgent`. Par exemple, l'implantation de la méthode `getTopology` permettant d'organiser les processus d'un cas de calcul en « anneau » est la suivante :

```
protected Topology getTopology(){
  return new Ring(size);
}
```

La méthode `getTopology` est invoquée lors de l'exécution de la méthode `execute` définie par un

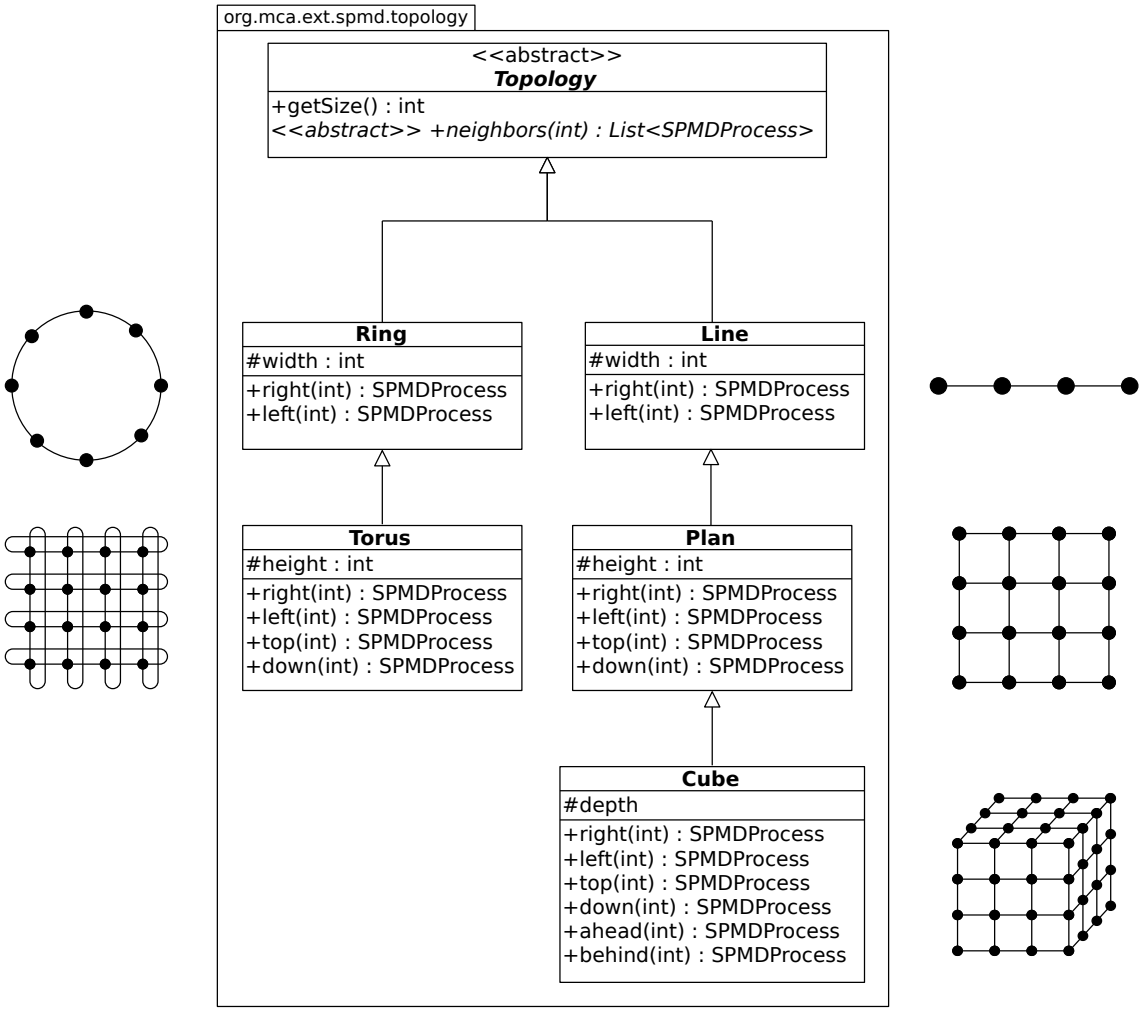


FIGURE 5.13 – Les différentes topologies et les classes correspondantes.

ComputeAgent de type `SPMDAgent`. Dans notre exemple, la classe `Ring` fournit les méthodes `right` et `left` pour récupérer respectivement une référence, de type `SPMDProcess`, vers le processus à droite et à gauche du processus *actif*, celui-ci étant fixé via le paramètre de type `int`. Le but étant d'utiliser ces méthodes dans l'implantation de la méthode `program` réalisée dans les implantations de la classe `SPMDAgent`. Par exemple, le code suivant permet de récupérer des données accessibles via un *DataHandler*, dont la propriété *name* est égale à « `column1` », se trouvant dans le *WorkSpace* (cf. Section 4.4.2) de l'agent *MCAWorker* exécutant le processus se trouvant à droite du processus *actif* :

```
SPMDProcess right = topology.right(rank);
DData data = right.recv("column1");
```

Notons que la méthode `recv` est bloquante, c'est à dire que le processus *actif* attend que le processus de droite mette à disposition les données en écrivant l'*entrée DataHandler* nommée `column1`. L'utilisation de la méthode `recv` demande une attention particulière. En effet, si lors d'un cas de

calcul, les processus échangent des données portant le même nom mais à des moments différents dans le code alors il est possible qu'un processus récupère des données qui ne soient pas celles désirées. Pour remédier à ce problème, il est possible de définir des barrières de synchronisation.

5.2.1.3 Barrière de synchronisation

L'utilisation d'une barrière de synchronisation est chose courante dans le monde du calcul distribué, comme par exemple MPI_Barrier dans *MPI*. En effet, il est souvent utile de définir un point spécifique dans le calcul où un processus doit attendre l'exécution d'autres processus avant de pouvoir continuer son exécution.

L'*API MCA-SPMD* propose deux types de barrière de synchronisation et sont implantées chacune par une méthode de la classe `SPMDAgent` (cf. Figure 5.12). Ces méthodes utilisent des *entrées* de type `Barrier`. Ce type d'*entrée* possède un attribut `name` qui est un identifiant unique pour la barrière.

Le développeur d'une classe héritant de la classe `SPMDAgent` peut utiliser :

- une *barrière totale* implantée par la méthode `barrier`. Ce type de barrière permet de synchroniser tous les processus d'un cas de calcul *SPMD*. Lors de l'utilisation de ce type de barrière, le premier agent *MCAWorker* exécutant le *ComputeAgent* qui atteint la barrière ajoute une *entrée* de type `Barrier` dans le *ComputationSpace* associé au cas de calcul. Il initialise la valeur de l'attribut `counter` à 1 et attend que la valeur de cet attribut soit égale au nombre de processus participant au cas de calcul pour continuer son exécution. Les *ComputeAgent* exécutés par les autres agents *MCAWorker* qui atteignent cette barrière récupèrent cette *entrée* de type `Barrier` et incrémente la valeur de l'attribut `counter` en invoquant le méthode `increment`. Ils attendent eux aussi que la valeur de `counter` soit égale au nombre de processus participant au cas de calcul pour continuer leur exécution.
- une *barrière de voisinage* implantée par la méthode `neighborBarrier`. Ce type de barrière permet à un agent *MCAWorker* de synchroniser le *ComputeAgent* qu'il exécute avec les *ComputeAgent* exécutés par les *MCAWorker*. La méthode `neighborBarrier` ajoute une *entrée* de type `Barrier` dans le *WorkSpace* local et invoque la méthode `barrier` définie dans la classe `SPMDProcess` sur chaque processus voisin. Cette méthode attend qu'une *entrée* de type `Barrier`, possédant une valeur identique à celle de l'attribut *name* de l'*entrée* écrite dans le *WorkSpace* local, soit ajoutée dans le *WorkSpace* de l'agent *MCAWorker* exécutant ce processus voisin.

5.2.2 Un exemple de calcul *SPMD* : l'équation de Laplace

La résolution d'Équations aux Dérivées Partielles (EDP) est un problème qui se pose dans la modélisation de nombreux phénomènes naturels. On rencontre, par exemple, ce type d'équation dans les problèmes de transfert de chaleur. Ce domaine est de plus en plus étudié dans le but d'essayer de diminuer les pertes de chaleur et ainsi économiser de l'énergie. Pour notre exemple,

le problème qui nous intéresse est celui de la distribution de la température sur une plaque carrée dont les bords sont soumis à une température constante.

En partant de l'équation de la chaleur [Leg56], il est possible, sous certaines conditions, d'écrire cette équation sous une forme plus simple telle que l'équation de Laplace [19] [Mai13]. Notre domaine d'étude, une plaque carrée, répond aux conditions car il s'agit d'un système fermé où la température tend vers un état stationnaire [20]. Le problème revient donc à résoudre l'équation de Laplace à deux dimensions telle que

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{5.1}$$

On considère que le problème est situé dans le plan xOy. Pour le résoudre, nous discrétisons la plaque en un maillage $\{(x_i, y_i)\}$ de pas h avec $h = x_{i+1} - x_i = y_{j+1} - y_j$ (cf. Figure 5.14(a)).

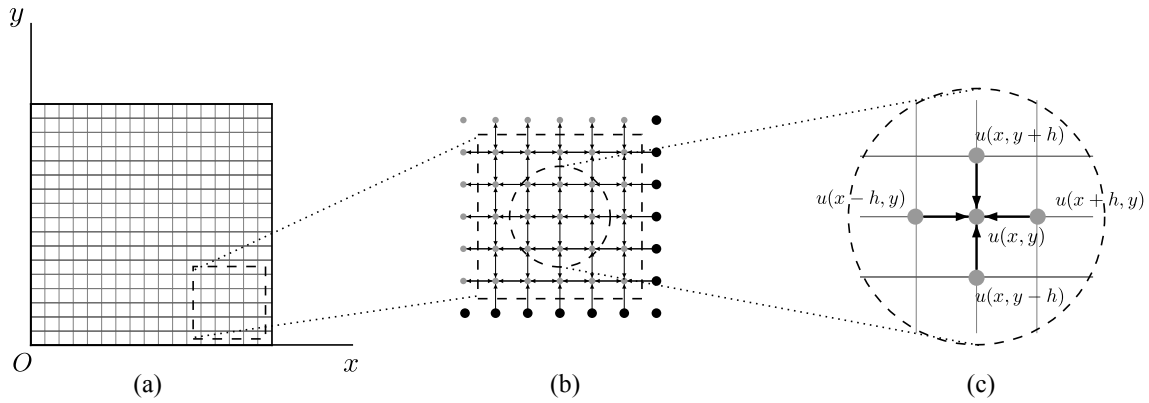


FIGURE 5.14 – Modélisation d'une plaque carrée dans le plan xOy : (a) maillage en 2 dimensions ; (b) communication entre les sommets à chaque itération ; (c) représentation d'un sommet $u(x, y)$ dans le plan xOy.

En utilisant la méthode des différences finies [All07], nous pouvons écrire les équations 5.2 et 5.3

$$\frac{\partial^2 u(x,y)}{\partial x^2} \equiv \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2} \tag{5.2}$$

$$\frac{\partial^2 u(x,y)}{\partial y^2} \equiv \frac{u(x,y+h) - 2u(x,y) + u(x,y-h)}{h^2} \tag{5.3}$$

En remplaçant les valeurs dans l'équation de Laplace (eq. 5.1), nous obtenons l'équation 5.4 :

$$u(x,y) = \frac{1}{4}\Big(u(x+h,y) + u(x-h,y) + u(x,y+h) + u(x,y-h)\Big) \tag{5.4}$$

Ainsi, on constate que la fonction u au point (x, y) est égale à la moyenne des 4 points adjacents

19. Pierre-Simon de Laplace (1749-1827) est un mathématicien, astronome et physicien français.
20. Un régime stationnaire est un processus physique qui est indépendant du temps

(cf. Figure 5.14(c)). Pour chaque point (x_i, y_j) du plan, nous pouvons donc écrire l'équation 5.4 (cf. Figure 5.14(b)).

5.2.2.1 Résolution séquentielle du problème

Pour résoudre ce problème nous modélisons la fonction u par une matrice U de nombres réels. Celle-ci est définie par le schéma itératif proposé par l'équation 5.5 où les indices i et j dénotent les points de l'espace discret.

$$U^{k+1}_{\mathrm{i,j}} = \frac{U^k_{\mathrm{i+1,j}} + U^k_{\mathrm{i-1,j}} + U^k_{\mathrm{i,j+1}} + U^k_{\mathrm{i,j-1}}}{4} \tag{5.5}$$

En imposant une température aux bords du domaine, le problème se résout en laissant évoluer la température jusqu'à la stabilisation de la dynamique. Si les conditions aux limites du domaine considéré sont de la forme de Dirichlet [21], nous pouvons en déduire l'algorithme 5.1.

Algorithme 5.1: Algorithme séquentiel de la résolution de l'équation de Laplace par l'itération de Jacobi.

Données :
$U_{n,n}$: matrice modélisant le domaine étudié
σ : précision souhaitée
T_{out} : Température imposée sur les bords
$MAXITER$: nombre maximal d'itérations

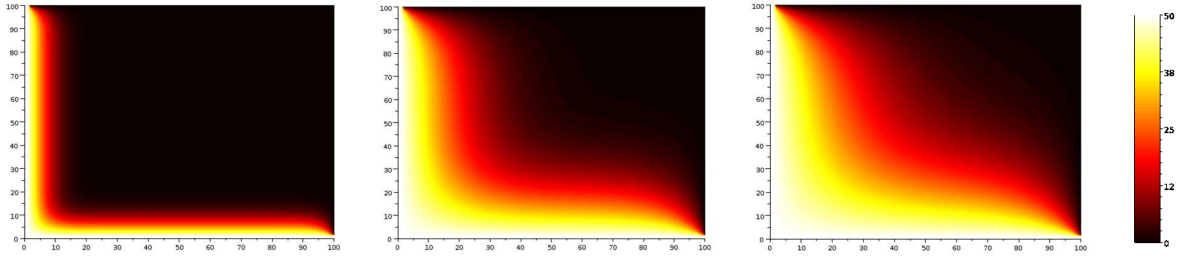
début
(1) **pour** $j = 1$ *à* n **faire**
  $U_{1,j} \longleftarrow T_{out}$
fin
(2) ...
$k \longleftarrow 0$;
$\varepsilon \longleftarrow \sigma + 1$;
(3) **tant que** $\varepsilon > \sigma$ *et* $k <$*MAXITER* **faire**
  pour $i = 2$ *à* $n - 1$ **faire**
    $\varepsilon \longleftarrow 0$ **pour** $j = 2$ *à* $n - 1$ **faire**
(4) $U^{k+1}_{i,j} \longleftarrow \left(U^k_{i+1,j} + U^k_{i-1,j} + U^k_{i,j+1} + U^k_{i,j-1}\right)/4$
(5) $\delta = |U^{k+1}_{i,j} - U^k_{i,j}|$
(6) $\varepsilon = max(\varepsilon, \delta)$
    fin
  fin
  $k \longleftarrow k + 1$
fin
fin

Les trois étapes principales de cet algorithme, appelé *itération de Jacobi*, sont les suivantes :

21. une condition aux limites de Dirichlet est imposée à une équation aux dérivées partielles lorsque l'on spécifie les valeurs que la solution doit vérifier sur les frontières

1. **Initialisation.** (1) On impose la valeur de T_{out} à tous les points de la frontière *Ouest*. (2) De la même manière, on impose cette valeur T_{out} à tous les points de la frontière *Sud* ($j = 1$) et la valeur 0 à tous les points des frontières *Nord* ($j = n$) et *Est* ($i = n$) .
2. **Itération.** (4) Pour chaque valeur $U_{i,j}$, on calcule la moyenne des valeurs voisines. (5) Ensuite, on calcule la valeur absolue de l'écart δ entre l'ancienne valeur et la nouvelle valeur. (6) Enfin, on calcule $\varepsilon = max(\varepsilon, \delta)$ afin de connaitre le plus grand écart pour une itération.
3. **Test (3).** Un seuil de précision σ est défini. L'itération se poursuit tant que $\varepsilon > \sigma$. Le processus converge vers la solution. Si le seuil de précision n'a pas été atteint, le processus fera au maximum MAXITER itérations.

Nous avons choisi d'étudier la distribution de la chaleur sur une plaque carrée de 2 mètres de coté soumis à une température de 50 ° C à ses bords Ouest et Sud. La plaque est donc modélisée en un maillage structuré (grille) en deux dimensions, chaque arête correspondant à 1 cm. Cette grille contient donc 40000 sommets (200×200). Comme nous l'avons vu dans la section précédente, nous pouvons transposer la fonction u, représentant la température en chaque point de la grille, en une matrice U de nombres réels. Notre matrice de travail est donc une matrice 200×200 avec $\forall i \in [\![1; 200]\!], U_{i,1} = 50$ et $\forall j \in [\![1; 200]\!], U_{n,j} = 50$, le reste des coefficients étant égal à 0. Nous avons implanté l'algorithme 5.1 sur ce domaine à l'aide de l'outil *Scilab* [Aff12]. Ceci nous permet de présenter la figure 5.15 qui illustre l'évolution de la température telle que nous pouvons la modéliser avec cet outil. Les différentes mesures ont été réalisées en faisant varier le nombre d'itération *MAXITER*.



(a) 100 itérations (b) 1000 itérations (c) 2000 itérations

FIGURE 5.15 – Visualisation à l'aide de l'outil *Scilab* de l'évolution de la température avec la résolution de l'équation de Laplace dans le plan en fonction du nombre d'itérations.

5.2.2.2 Application à la plate-forme de calcul *MCA*

L'algorithme 5.1 a un fort potentiel de parallélisation. Une stratégie consiste à partager la matrice U en sous-matrices (cf. Figure 5.16a) et d'en attribuer une à chaque processus participant au calcul (cf. Figure 5.16b). Suivant une approche *SPMD* (cf. Section 1.1.3.1), chaque processus exécute cet algorithme sur la sous-matrice qui lui est attribuée. Pour calculer les valeurs se

trouvant au bord de sa sous-matrice, un processus doit connaître les valeurs des bords des sous-matrices voisines. A chaque itération, tous les processus communiquent donc les valeurs des bords de leur sous-matrice locale aux processus traitant les sous-matrices voisines.

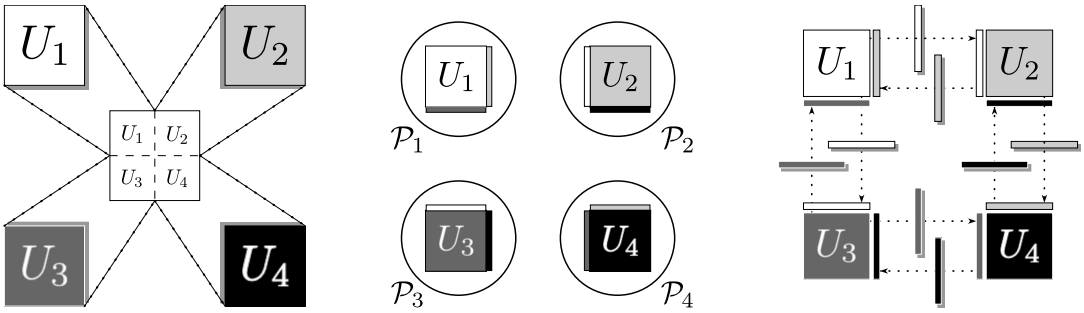


FIGURE 5.16 – Modélisation d'une itération de Jacobi parallèle sur 4 processus : (a) partage de la matrice en sous-matrices (U_1, U_2, U_3 et U_4) ; (b) chaque processus ($\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ et $\mathcal{P}_4$) reçoit une sous-matrice ainsi que les bords des sous-matrices voisines. (c) : à chaque itération les bords sont mis-à-jour et sont échangés avec les processus qui traitent les sous-matrices voisines.

Pour déployer un cas de calcul suivant le modèle *SPMD* sur la plate-forme *MCA*, le développeur doit configurer une instance de la classe `SPMDCaseDeployer` (cf. Figure 5.10) dans le descripteur de déploiement :

```
1 <spmd-deployer numberOfProcesses="12">
2   <spmd-program ref="jacobiAgent" />
3 </spmd-deployer>
```

L'attribut `numberOfProcesses` de la balise `<spmd-deployer />` définit le nombre de processus nécessaires pour le cas de calcul et la balise `<spmd-program />` définit quel est le *ComputeAgent* qui sera exécuté par chaque agent *MCAWorker* participant au cas de calcul. Ce *ComputeAgent* est une instance de la classe `JacobiAgent` qui hérite de la classe `SPMDAgent` (cf. Figure 5.12). L'exécution de ce *ComputeAgent* demande la définition de propriétés globales comme le nombre maximal d'itérations (`MAXITER`), la précision souhaitée (`THRESHOLD`) et les valeurs aux bords du domaine étudié (`BORDER_SOUTH`, `BORDER_NORTH`, `BORDER_EAST`, `BORDER_WEST`).

```
1 <properties>
2   <property key="MAXITER" value="5000" />
3   <property key="THRESHOLD" value="0.0000001" />
4   <property key="BORDER_SOUTH" value="50" />
5   <property key="BORDER_EAST" value="0" />
6   <property key="BORDER_NORTH" value="0" />
7   <property key="BORDER_WEST" value="50" />
8 </properties>
```

Enfin, le domaine étudié est représenté par une matrice distribuée (cf. Section 4.4.4.2). Celle-ci

est déclarée dans le descripteur de déploiement par la balise `<matrix />`. Les attributs `rowSize` et `columnSize` fixent les dimensions de la matrice alors que les attributs `rowPartSize` et `columnPartSize` fixent les dimensions d'une partie de cette matrice, modélisée par une instance de la classe `DMatrixPart` (cf. Figure 4.28). La balise `<datahandler-factory />` déclare une fabrique d'*entrée* de type *DataHandler* (cf. Section 4.4.1.2) qui sont utilisées pour lire et écrire les différentes parties de cette matrice distribuée. Dans notre cas précis, nous utilisons des *entrées DataHandler* de type `LocalDataHandler` car tous les agents *MCAWorker* partagent le répertoire `/home/mca/`. Celui-ci est synchronisé sur chaque nœud de la grille grâce à l'utilisation du service *NFS* (cf. Section 5.1.1).

```
<data>
    <matrix name="input" rowSize="200" columnSize="200" rowPartSize="200" columnPartSize="200">
      <datahandler-factory class="org.mca.entry.LocalDataHandlerFactory">
        <properties>
        <property name="path" value="/home/mca/cases/jacobi/" />
        </properties>
      </datahandler-factory>
    </matrix>
</data>
```

Cette matrice contenant les données de description du maillage respecte les règles d'accès vues dans la chapitre 4. Les propriétés de symétrie de cette matrice sont utilisées pour la gestion des données par les *ComputeAgent* (cf. Figure 5.16c).

5.2.3 Évaluation du surcoût introduit par l'adaptabilité offerte par le framework *MCA*

Dans cette section nous souhaitons évaluer la capacité d'adaptabilité offerte par l'utilisation de notre framework. Le graphique de la figure 5.17 présente les résultats d'une première expérience. Celle-ci consiste à évaluer les performances de la plate-forme en fonction de la taille des données qui sont traitées par le *ComputeAgent* exécuté par les agents *MCAWorker*. La taille des données traitées est proportionnelle à la taille des données échangées entres les agents *MCAWorker* voisins. Nous remarquons que l'impact des échanges est faible. En effet, la courbe suit une trajectoire qui montre que le temps d'exécution augmente proportionnellement à la taille des données manipulées par tous les agents *MCAWorker*. Il n'y a donc pas de surcoût dû au transfert de données entre les agents *MCAWorker* voisins.

De son coté, la figure 5.18 présente les résultats obtenus en réalisant une itération de Jacobi dans laquelle chaque *ComputeAgent* traite une sous-matrice de 2000×2000 doubles qu'il génère au début de son exécution. Nous avons réalisé la résolution de ce cas de calcul à plusieurs reprises en incrémentant à chaque exécution le nombre d'agents *MCAWorker* connectés à la plate-forme *MCA*. L'ajout d'un agent *MCAWorker* entraine alors l'augmentation de la taille totale des

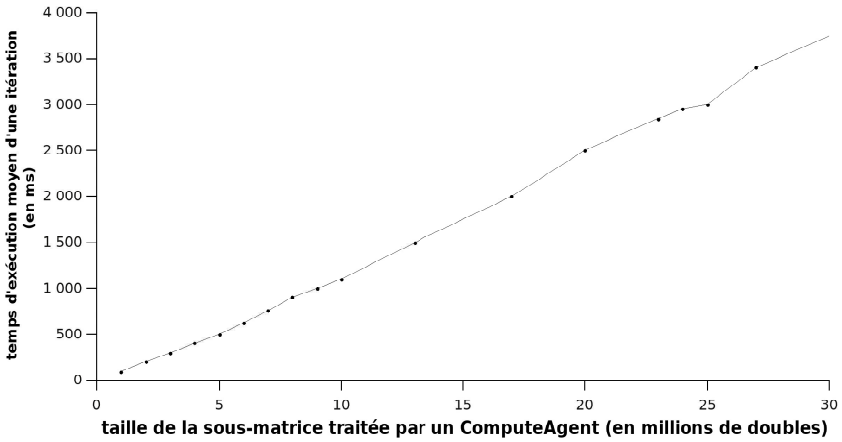


FIGURE 5.17 – Surcoût induit par l'utilisation du framework *MCA*

données traitées de 2000×2000 doubles. Notre objectif étant de montrer que l'augmentation de la taille de données traitées par l'ensemble des agents *MCAWorker* n'impacte pas, ou alors très peu, le temps nécessaire à la résolution du cas de calcul si on ajoute des agents *MCAWorker* au réseau. L'analyse de ces résultats montre que la plate-forme répond à l'augmentation du nombre d'agents *MCAWorker* sans augmenter le temps d'exécution du cas de calcul. En effet, le temps d'exécution moyen d'une itération reste stable malgré l'augmentation du nombre d'agents *MCAWorker* et donc, par conséquent, de la taille des données traitées. Cela nous permet de dire que l'auto-adaptation de notre plate-forme aux nombres de ressources disponibles est performante.

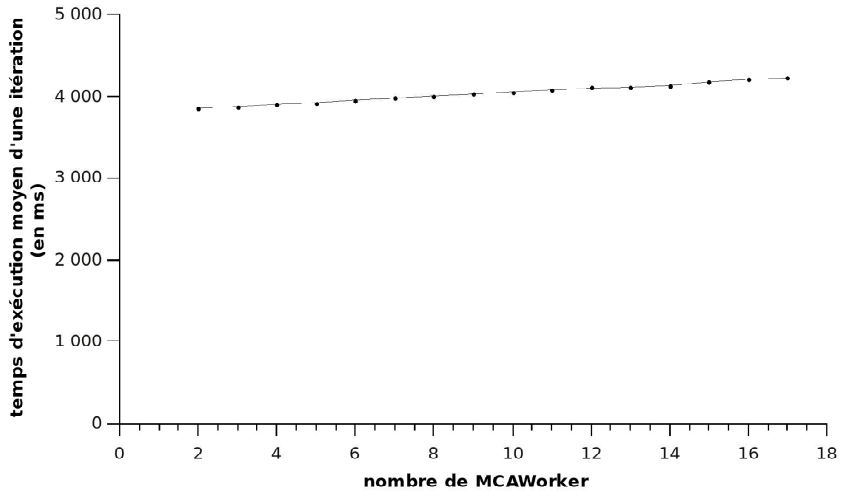


FIGURE 5.18 – Adaptabilité de la plate-forme MCA en fonction du nombre d'agents *MCAWorker*

5.3 Évaluation de la tolérance aux pannes fournie par le framework *MCA*

Après avoir évaluer l'adaptabilité du framework *MCA*, nous souhaitons à présent évaluer la tolérance aux pannes offerte à tout cas de calcul s'exécutant sur la plate-forme *MCA*. Pour réaliser cette évaluation, nous avons choisi un cas de calcul de type *Maître-Travailleurs* avec l'approximation du nombre π par une méthode de *Monte Carlo* (Section 5.3.1). A partir de ce cas de calcul, nous réalisons différentes mesures pour le surcoût introduit par la tolérance aux pannes présente dans notre framework avec la présence ou non de panne (Section 5.3.2).

5.3.1 Approximation du nombre π par une méthode de *Monte Carlo*

Les *méthodes de Monte Carlo* forment une classe d'algorithmes qui reposent sur la répétition d'échantillons de valeurs aléatoires afin d'approcher une valeur numérique. Ces méthodes sont utilisées dans différents domaines comme la finance, la physique ou les mathématiques. L'application de ces méthodes varie selon les domaines mais l'utilisation de nombres aléatoires pour résoudre statistiquement un problème est présente dans chacune d'entre elles.

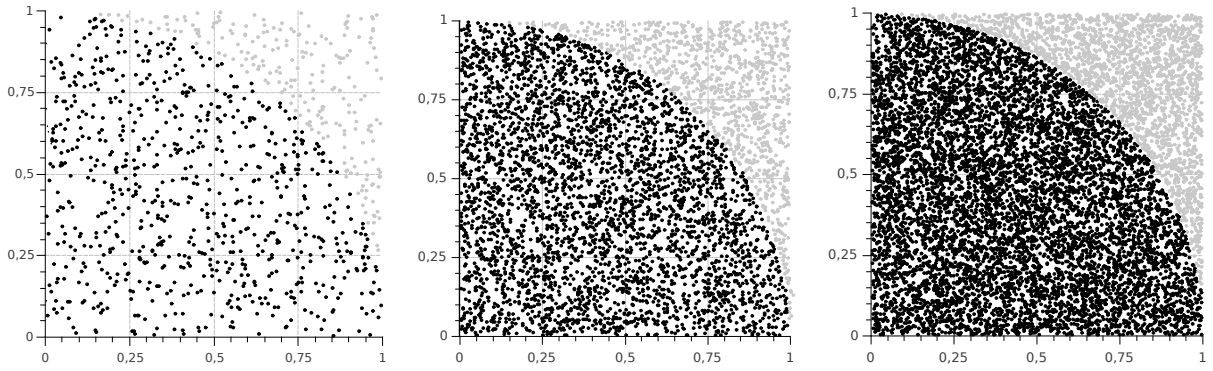
L'algorithme pour approcher la valeur du nombre π consiste à tirer de façon aléatoire un nombre N de couple (x, y), avec x et y des valeurs comprises entre 0 et 1. Le point M de coordonnées (x, y) appartient au disque de centre $(0, 0)$ de rayon 1 si et seulement si $x^2+y^2 \leq 1$. La probabilité pour que le point M appartienne au disque est de $\pi/4$. Si n est le nombre de points $M(x, y)$ dans le disque de rayon 1 alors il est possible d'approximer la valeur du nombre π par le rapport suivant (eq. 5.6) :

$$\pi = 4 \times \frac{n}{N} \tag{5.6}$$

Une version distribuée de cet algorithme consiste à utiliser le paradigme *Maître-Travailleurs*. Dans ce cas, le processus *maître* génère des points de type $M(x, y)$ de façon aléatoire. Les processus *travailleurs* vérifient si ces points appartiennent au disque de rayon 1 et de centre $(0, 0)$ et renvoient une valeur booléenne en fonction du résultat. Le processus *maître* récupère alors les résultats retournés par les processus *travailleurs* et calcule le rapport défini par l'équation 5.6.

5.3.1.1 Implantation avec le framework *MCA*

Notre framework *MCA* propose une *API* (package `org.mca.ext.masterworker` de la figure 5.20) qui facilite la mise en place de cas de calcul suivant le modèle *Maître-Travailleurs*. Pour résoudre un cas de calcul qui suit ce modèle, le framework *MCA* impose au développeur d'implanter deux nouvelles classes afin de définir deux nouveaux types de *ComputeAgent* :



(a) 1000 points (b) 10000 points (c) 100000 points

FIGURE 5.19 – Visualisation de l'évolution d'une simulation de *Monte Carlo* en fonction du nombre de points testés.

– une classe héritant de la classe `MasterAgent` pour définir le *ComputeAgent* responsable d'exécuter le code du processus maître. Ce *ComputeAgent* est associé à une *entrée* de type `MasterTask`. Il n'y a qu'une seule *entrée* de ce type dans le *ComputationSpace* du cas de calcul. Cette *entrée* est ajoutée lors du déploiement du cas de calcul par l'implantation de la méthode `deployTasks` de la classe `MWCaseDeployer`. L'agent *MCAWorker* qui récupère cette *entrée* est alors considéré comme le processus *maître*.
– une classe héritant de la classe `AbstractComputeAgent` pour définir le *ComputeAgent* responsable d'exécuter le code d'un processus *travailleur*. Ce *ComputeAgent* est associé à des *entrées* de type `WorkerTask`. Ces *entrées* sont ajoutées au *ComputationSpace* par l'agent *MCAWorker* qui exécute le processus *maître* (via le *ComputeAgent* de type `MasterAgent`). L'agent *MCAWorker* qui récupère une de ces *entrées* est alors considéré comme un processus *travailleur*.

Le squelette d'un processus maître est donc fixé par la classe abstraite `MasterAgent` (cf. Figure 5.20). Cette classe définit trois méthodes abstraites `startMasterProcess`, `performResult` et `performPostTreatment`. La figure 5.21 présente le diagramme de séquence lors de l'exécution d'un *ComputeAgent* modélisant un processus maître. Ce type d'agent exécute en parallèle deux processus légers (*thread*) :

– un processus défini par la classe `TaskDeployer`. Ce processus invoque la méthode `startMasterProcess` qui a pour but de créer les *entrées* de type `WorkerTask` et de les ajouter dans le *ComputationSpace* à l'aide de la méthode `addWorkerTask`. Toutes les *entrées* `WorkerTask` ajoutées ont la même valeur pour leur attribut `computeAgentUrl`, c'est à dire la valeur de l'attribut `workerAgentUrl` de l'agent `MasterAgent` ;
– un processus défini par la classe `ResultCollector`. Ce processus récupère par lot des *entrées Task* de type `WorkerTask` qui sont dans l'état DONE (cf. Figure 4.19) et analyse leurs résultats dans la méthode `performResults`. Cette méthode renvoie une valeur booléenne pour définir si la collecte des tâches doit se finir ou non.

Dans le cas où la méthode `performResults` retourne la valeur `true`, le *ComputeAgent* signale la fin du cas de calcul et invoque la méthode `performPostTreatment` pour finaliser l'exécution du processus

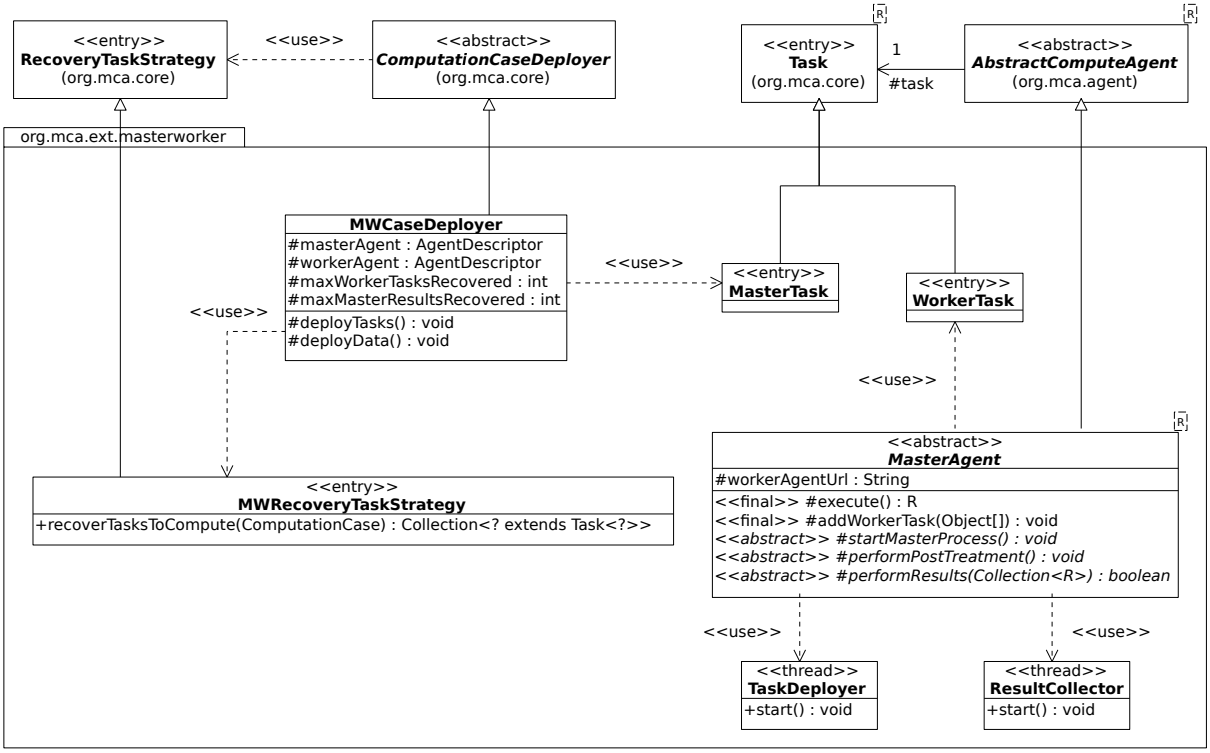


FIGURE 5.20 – Diagramme de classes de l'*API* permettant la résolution de cas de calcul suivant le modèle *Maitre-Travailleurs*.

maître.

Notons la présence de la classe `MWRecoveryTaskStrategy` qui propose une sous-classe de *RecoveryTaskStrategy* (cf. Section 4.4.1.2). Cette classe fixe la stratégie utilisée par les agents *MCAWorker* pour récupérer les différentes tâches à traiter. Elle consiste à récupérer en priorité l'*entrée Task* de type `MasterTask` avant de récupérer `maxWorkerTasksRecovered` *entrées Task* de type `WorkerTask`. Cette stratégie permet d'être sûr que le processus maître est toujours exécuté si un agent *MCAWorker* est disponible. En effet, si l'agent *MCAWorker* qui exécute le *ComputeAgent* de type `MasterAgent` tombe en panne, le prochain agent *MCAWorker* qui demande des tâches à traiter reçoit la tâche qui correspond au processus maître.

Le déploiement d'un cas de calcul suivant ce modèle est réalisé par une instance de la classe `MWCaseDeployer` (cf. Figure 5.20). Cette instance est paramétrée dans le descripteur de déploiement (cf. Section 5.1.3.1) par la balise `<mw-deployer />` (cf. Figure 5.7) qui possède deux attributs :

- `maxWorkerTasksRecovered` qui détermine la taille maximal des lots de tâches de type `WorkerTask` récupérées par les agents *MCAWorker*. Cette valeur permet de configurer l'instance de la classe `MWRecoveryTaskStrategy`.
- `maxMasterResultsRecovered` qui détermine la taille maximal des lots de résultats récoltés par le processus *maître* à l'aide de l'instance de type `ResultCollector`.

Les balises `<master />` et `<worker />` fixent respectivement les agents de type *ComputeAgent* définissant le processus *maître* et les processus *travailleurs*. L'attribut `ref` fait référence à l'attribut

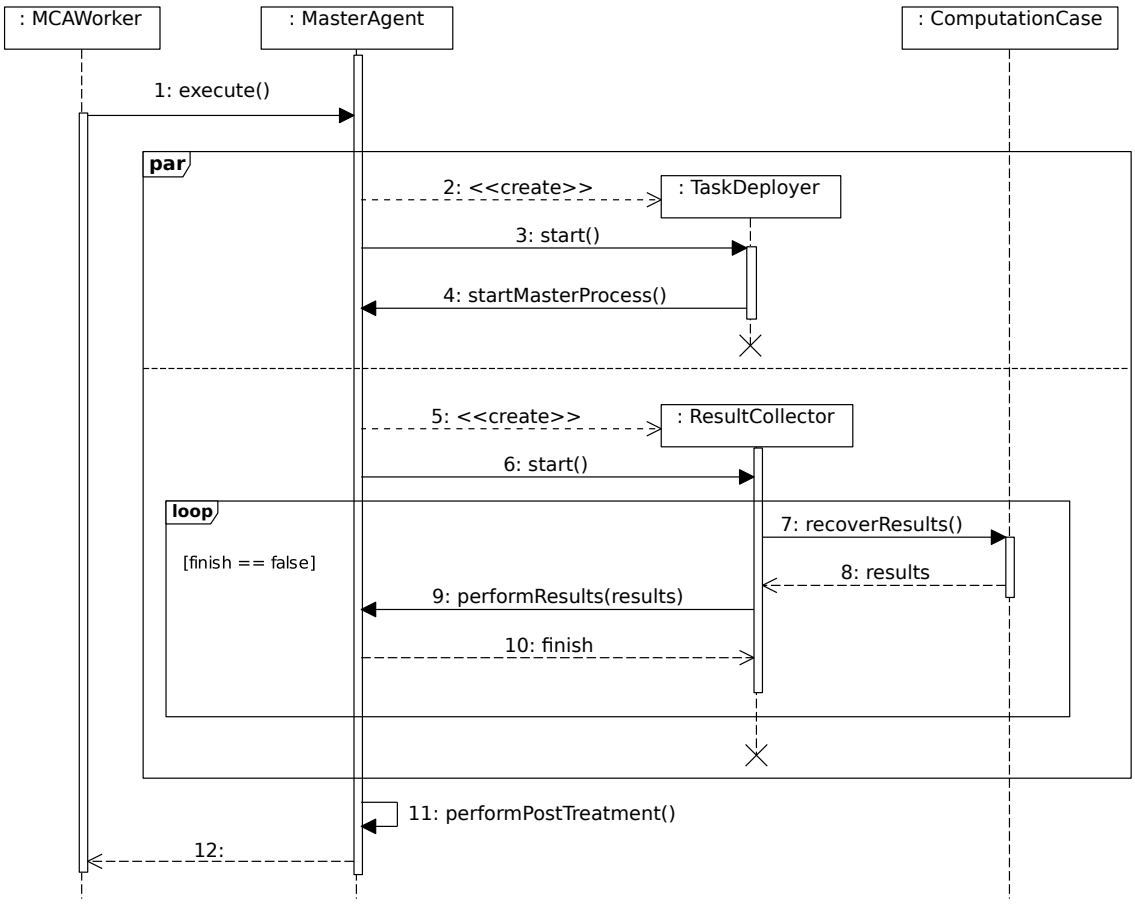


FIGURE 5.21 – Diagramme de séquence de l'exécution générique d'un agent de type *MasterAgent*.

`name` de la balise `<agent />` (cf. Figure 5.9) utilisée lors de la configuration d'un *ComputeAgent*.

```
1  <case name="PI">
2    <description>Calcul du nombre Pi par une méthode de Monte Carlo</description>
3      <properties>
4        <property key="NB_SHOTS" value="100000"/>
5      </properties>
6      <mw-deployer maxWorkerTasksRecovered="100" maxMasterResultsRecovered="1000">
7        <master ref="masterAgent" />
8        <worker ref="workerAgent" />
9      </mw-deployer>
10 </case>
```

FIGURE 5.22 – Extrait du descripteur de déploiement du cas de calcul pour l'approximation du nombre π.

Pour l'implantation de l'algorithme présenté dans la section 5.3.1, nous avons développé deux nouvelles classes : la classe `PIMasterAgent` (cf. Figure 5.24) pour le *ComputeAgent* modélisant le processus *maître* et la classe `PIWorkerAgent` (cf. Figure 5.23) pour le *ComputeAgent* modélisant un processus *travailleur*.

La classe `PIMasterAgent` (cf. Figure 5.24) qui hérite de la classe `MasterAgent` (cf. Figure 5.20) implante les trois méthodes déclarées abstraites. Dans la méthode `startMasterProcess` (l.7 à 15), l'instance de `PiMasterAgent` ajoute `NB_SHOTS` (valeur de la propriété globale définie dans le descripteur de déploiement de la figure 5.22) *entrées Task* (de type `WorkerTask`) au *ComputationSpace* associé au cas de calcul. Chaque tâche ajoutée possède deux paramètres qui correspondent aux coordonnées d'un point du plan générées de façon aléatoire via la méthode `Math.random`. La méthode `performResults` (l. 17 à 23) comptabilise le nombre de résultats des tâches dont la valeur est égale à `true`, ce qui signifie que le point correspondant à la tâche appartient au disque de rayon 1 de centre $(0, 0)$. Ce traitement est réalisé tant que le nombre de résultats récupérés est inférieur à `NB_SHOTS`. La méthode `performPostTreatment` (l. 25 à 28) calcule une approximation de la valeur de π avec le rapport défini par l'équation 5.6 et publie cette valeur en ajoutant une *entrée* de type *Property* au *ComputationSpace*.

```
1  package org.mca.example.pi.agent;
2
3  public class PIWorkerAgent extends AbstractComputeAgent<Boolean> {
4   protected Boolean execute() throws Exception {
5    double x = task.getDoubleParameter(0);
6    double y = task.getDoubleParameter(1);
7    return Math.hypot(x, y) < 1;
8   }
9  }
```

FIGURE 5.23 – Implantation du *ComputeAgent* définissant un processus *travailleur* lors de l'approximation du nombre π.

La classe `PiWorkerAgent`, qui est un sous-type de la classe `AbstractComputeAgent` (cf. Figure 4.23), définit le code d'un processus *travailleur* lors de l'approximation du nombre π. L'implantation de la méthode `execute` consiste à lire la valeur des deux paramètres de l'*entrée* de type `WorkerTask`, ce qui correspond aux coordonnées (x, y) d'un point du plan, et à renvoyer le résultat du test $x^2 + y^2 \leq 1$.

5.3.2 Évaluation du coût introduit par la tolérance aux pannes

Pour évaluer la tolérance aux pannes fournie par notre framework *MCA*, nous commençons par comparer uniquement les temps d'exécution lors de l'accès à un *ComputationSpace*. Les premiers résultats, présentés par les tableaux 5.1, 5.2 et 5.3 figure **??**, ont été obtenus respectivement lors de l'exécution d'une action de lecture (*read*), d'écrire (*write*) et de récupération (*take*) en

```
1  package org.mca.example.pi.agent;
2
3  public final class PIMasterAgent extends MasterAgent<Boolean> {
4
5   private int numberOfShots;
6
7   protected void startMasterProcess() {
8    numberOfShots = Integer.valueOf(properties.get("NB_SHOTS"));
9    for (int i = 0; i < numberOfShots; i++) {
10    double x = Math.random();
11    double y = Math.random();
12    Object[] parameters = new Object[]{x,y};
13    addWorkerTask(parameters);
14   }
15  }
16
17  protected boolean performResults(Collection<Boolean> results) {
18   for (Boolean result : results) {
19    if (result) nbShotsIn++;
20   }
21   if (nbRecoveredResult == numberOfShots) return true;
22   return false;
23  }
24
25  protected void performPostTreatment() throws Exception {
26   double result = 4 * nbShotsIn / numberOfShots;
27   computationCase.addProperty("RESULT", result);
28  }
29 }
```

FIGURE 5.24 – Implantation du *ComputeAgent* définissant le processus *maître* lors de l'approximation du nombre π.

fonction de la configuration de tolérance aux pannes sur le *ComputationSpace*. Les différentes configurations possibles sont présentées dans la section 4.3.3.

Le résultat suivant, présenté par le tableau 5.4, permet d'évaluer l'impact de l'utilisation de notre framework, avec ou sans tolérance aux pannes. La colonne libellée « Sans *MCA* » présente les résultats obtenus sans l'utilisation de notre framework. Dans ce cas, nous avons simplement démarré le service *Outrigger* fourni par Jini™ (cf. Section 4.2.1).

Nous avons réalisé le cas de calcul présenté dans la section 5.3.1 et nous avons fait varier le nombre d'agent *MCAWorker* démarrés au départ de l'exécution, c'est à dire que le nombre d'agents *MCAWorker* ne varie pas au cours de la résolution du cas de calcul. Le tableau 5.5 présente les résultats obtenus lors de la résolution du cas de calcul en fonction du nombre d'agents *MCAWorker* disponibles sur le réseau.

Le tableau 5.5 nous permet d'évaluer l'impact de l'utilisation de notre framework. Nous indiquons aussi les temps d'exécution en fonction de l'activation ou non de la tolérance aux pannes (FT). L'analyse de ces résultats montre que l'utilisation de notre framework augmente le temps de calcul par rapport à une exécution dans cette couche logicielle. L'augmentation du nombre de

Mode \ Topologie	*Actif-Passif*	*Actif-Actif*
Synchrone	0.0018	0.0018
Asynchrone	0.0018	0.0018

TABLE 5.1 – Temps d'exécution d'une action *read* sur un *ComputationSpace* en fonction de sa configuration de tolérance aux pannes

Mode \ Topologie	*Actif-Passif*	*Actif-Actif*
Synchrone	0.002	0.0019
Asynchrone	0.0018	0.0017

TABLE 5.2 – Temps d'exécution d'une action *write* sur un *ComputationSpace* en fonction de sa configuration de tolérance aux pannes

Mode \ Topologie	*Actif-Passif*	*Actif-Actif*
Synchrone	0.0021	0.002
Asynchrone	0.0018	0.0017

TABLE 5.3 – Temps d'exécution d'une action *take* sur un *ComputationSpace* en fonction de sa configuration de tolérance aux pannes

	Exécutions (en ms)		
	Sans *MCA*	*MCA* sans FT	*MCA* avec FT
Read	0.0015	0.0018	0.0018
Write	0.001	0.002	0.0045
Take	0.0015	0.0018	0.0047

TABLE 5.4 – Impact de l'utilisation du framework *MCA* sur les actions *read*, *write* et *take*.

Nombre de *MCAWorker*	Exécutions (en ms)		
	Sans *MCA*	*MCA* sans FT	*MCA* avec FT
5	2027	2203	2500
10	1025	1201	1467
15	522	612	752

TABLE 5.5 – Impact de l'utilisation du framework *MCA* lors de la résolution d'un cas de calcul.

MCAWorker tend à réduire cette différence de temps d'exécution. L'activation de la tolérance aux pannes augmente légèrement le temps d'exécution et cette augmentation tend aussi à se réduire avec l'augmentation de nombre de *MCAWorker*.

5.4 *MCA-Skel* : une *API* pour l'utilisation de squelettes algorithmiques

De nombreux algorithmes parallèles peuvent être caractérisés et classés en un petit nombre de schémas génériques de calcul. La programmation basée sur les squelettes algorithmiques data-parallèle, introduit par [Col89]), rend ces schémas abstraits et fournit au programmeur une trousse à outils dans laquelle les spécifications s'affranchissent des variations d'architecture. Des travaux ont déjà été réalisés dans ce domaine de recherche au sein de notre laboratoire [LGH12]. La simplicité est une des forces de ce type de programmation et l'implantation que nous avons réalisée, via l'*API MCA-Skel* (section 5.4.1), permet à un programmeur de garder cette simplicité dans la mise en œuvre de cas de calcul utilisant ces squelettes algorithmiques sur la plate-forme *MCA*. Nous réalisons une *transformée de Fourier rapide* via une composition de squelettes algorithmiques comme exemple d'application de l'*API MCA-Skel* (section 5.4.2). Notons que l'objectif de cette section est de montrer la simplicité d'utilisation de notre framework *MCA*, il n'y a donc pas de résultats fournis sur les temps d'exécution mais uniquement la présentation du code source développé pour la résolution du cas de calcul exemple.

5.4.1 Implantations de squelettes algorithmiques

Nous avons choisi de présenter dans cette section les squelettes algorithmiques utilisés dans la section 5.4.2 avec notre exemple d'application : c'est à dire les squelettes *repl*, *map*, *mapidx*, *reduce*, *scan* et *dh*. Tous ces squelettes sont des squelettes data-parallèle et s'appliquent sur des vecteurs de données distribuées sur différents processus. Une implantation de chacun des ces squelettes a été réalisée à l'aide du framework *MCA*. Chaque squelette est associé à une méthode de la classe `SkeletonMasterAgent` (cf. Figure 5.25) et un *ComputeAgent* défini par une classe héritant de la classe `SkeletonAgent`. L'*API MCA-Skel* propose un ensemble de classes facilitant la résolution de cas de calcul maniant des squelettes algorithmiques. Un cas de calcul de ce type doit définir une tâche principale associée à un *ComputeAgent* de type `SkeletonMasterAgent`. Ce type de *ComputeAgent* agit de façon identique à un agent de type `MasterAgent` (cf. Section 5.3.1.1). Son rôle est d'organiser l'enchainement des différents squelettes algorithmiques utilisés par le cas de calcul en invoquant les méthodes correspondantes.

Notons, comme le suggère le figure 5.25, que la classe `SkeletonAgent` hérite de la classe `SPMDAgent`. En effet, la résolution d'un squelette sur notre plate-forme est comparable à la résolution d'un cas de calcul suivant le modèle SPMD (cf .5.2). Ici, les données manipulées par cette *API* sont

définies par des *SDD* (cf. Section 4.4.4) de type `DVector`. L'exécution d'un squelette est alors réalisée par n tâches via l'ajout de n *entrées* de type `SkeletonTask` dans le *ComputationSpace* associé au cas de calcul. Chaque *entrée Task* déclare le même *ComputeAgent*, c'est à dire la même valeur pour la propriété `computeAgentUrl`. Le nombre n de tâches est égal au nombre de parties du vecteur d'entrée sur lequel s'applique le squelette. Chaque tâche contient un entier (avec l'attribut `part`) correspondant à une partie de ce vecteur.

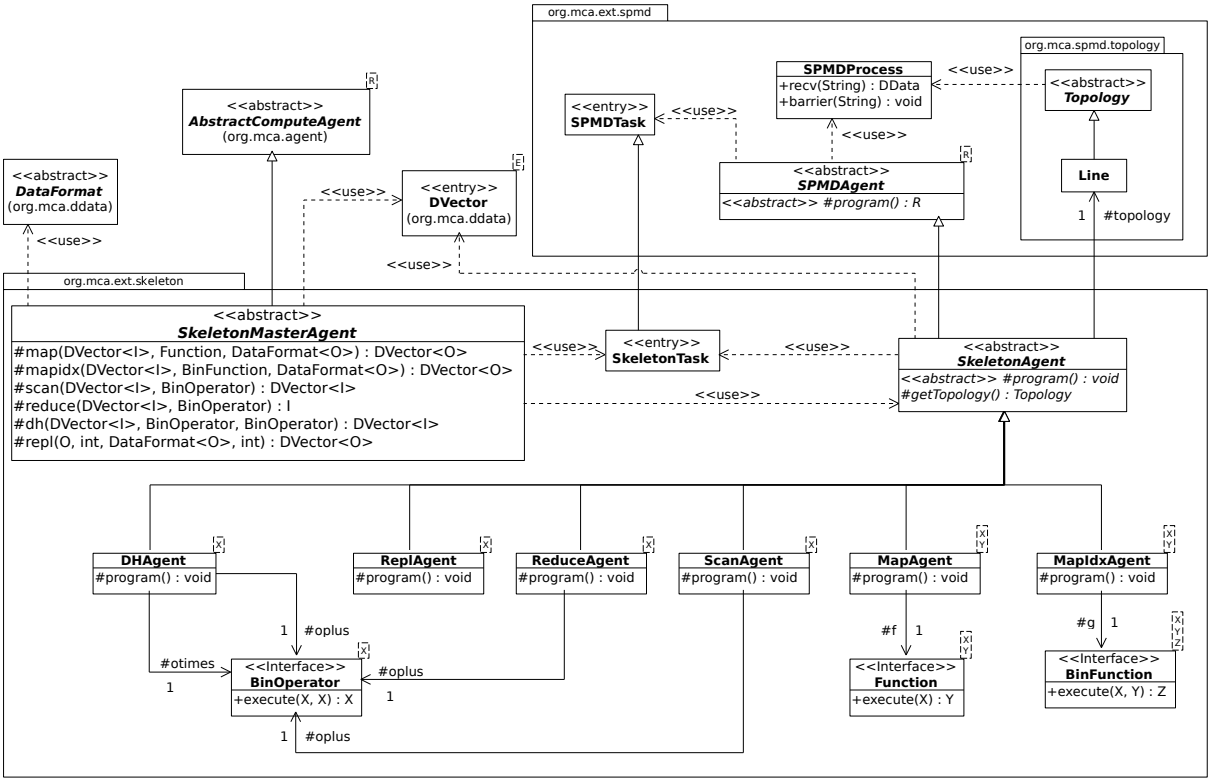


FIGURE 5.25 – Diagramme de classes de l'*API MCA-Skel*.

Nous présentons dans la suite de cette section les différents squelettes implantés dans l'*API MCA-Skel*. Pour chaque squelette, nous donnons sa définition et la méthode correspondante dans notre *API*. Toutes ces méthodes sont définies dans la classe `SkeletonMasterAgent`.

5.4.1.1 Réplication

Le premier squelette que nous présentons est le squelette **repl**. Celui-ci crée une nouvelle liste contenant n fois l'élément x tel que

$$x : repl(x, n) = [x, \ldots, x]$$

Pour exécuter ce squelette, l'*API MCA-Skel* propose la méthode `repl` :

- `DVector<O> repl(O value, int n, DataFormat<O> format, int partSize)`

Cette méthode ajoute une nouvelle *SDD* dans le *ComputationSpace*. La *SDD* est de type `DVector` et contient `n` fois la valeur `value` de type `O`. Chaque partie de la *SDD* contient `partSize` éléments. Le format pour lire ou écrire les éléments est donné par le paramètre `format` qui est une instance d'une sous-classe de `DataFormat` (cf. Section 4.4.4).

5.4.1.2 Map

Le squelette **map** applique une fonction unitaire, notée f, à tous les éléments d'une liste :

$$map(f, [x_1, \ldots, x_n]) = [f(x_1), \ldots, f(x_n)]$$

Ce squelette est équivalent au modèle *SPMD* où un programme unique, ici représenté par la fonction f, est exécuté sur un ensemble de données en parallèle. Pour exécuter ce squelette, l'*API MCA-Skel* propose la méthode `map` :

- `DVector<O> map(DVector<I> input, Function<I, O> f, DataFormat<O> format)`

Si n est le nombre de parties de la *SDD* `input`, alors la méthode `map` ajoute n tâches, de type `SkeletonTask`, dans le *ComputationSpace* associé au cas de calcul. Chaque tâche définit le même *ComputeAgent* de type `MapAgent`. Les agents *MCAWorker* qui traitent les tâches associées à l'exécution de ce squelette travaillent uniquement sur la partie des données qu'ils ont chargée en local et aucune communication n'est nécessaire entre les agents *MCAWorker*.

Le squelette **mapidx**, variante du squelette *map*, applique une fonction binaire g à chaque élément de la liste d'entrée. La fonction g prend alors en paramètre un élément de la liste et son indice :

$$mapidx(g, [x_1, \ldots, x_n]) = [g(1, x_1), g(2, x_2), \ldots, g(n, x_n)] \tag{5.7}$$

L'implantation de ce squelette est très proche de celle du squelette *map* : la méthode `mapidx` utilise des agents de type `MapIdxAgent` avec une instance de la classe `BinFunction` comme paramètre pour définir la fonction binaire g :

- `DVector<O> mapidx(DVector<I> input, BinFunction<Integer, I ,O> g, DataFormat<O> format)`

5.4.1.3 Opérations collectives

Les deux squelettes suivants (**reduce** et **scan**) sont fournis en tant qu'opérations collectives dans le standard MPI (respectivement *MPI_ Reduce* et *MPI_ Scan*). Le squelette **reduce** combine tous les éléments de la liste d'entrée en utilisant une opération binaire associative $\oplus$:

$$reduce(\oplus, [x_1, \ldots, x_n]) = x_1 \oplus \cdots \oplus x_n$$

Le squelette **scan** calcule les réductions partielles de tous les éléments de la liste en parcourant la liste de gauche à droite et combine les éléments avec l'opérateur binaire associatif $\oplus$ tel que

$$scan(\oplus, [x_1, \ldots, x_n]) = [x_1, (x_1 \oplus x_2), \ldots, ((x_1 \oplus x_2) \cdots \oplus x_n)]$$

L'implantation de ces deux squelettes est réalisée par les méthodes suivantes :

- `I reduce(DVector<I> input, BinOperator<I> oplus)`
- `DVector<I> scan(DVector<I> input, BinOperator<I> oplus)`

Dans les deux cas, des communications entre les différents *MCAWorker* sont nécessaires pour partager les données du vecteur d'entrée. Ces communications montrent l'importance des *SDD* dans le traitement de ce type de squelettes.

5.4.1.4 Homomorphisme distribuable

Le squelette homomorphisme distribuable, noté **dh**, est un squelette plus complexe. Il est utilisé pour exprimer une classe spécifique de problème, « diviser pour régner » tel que

$$dh(\oplus, \otimes, [x_1, \ldots, x_n]) = [y_1, \ldots, y_n]$$

Ce squelette transforme une liste $[x_1, \ldots, x_n]$ de taille 2^m en une liste $[y_1, \ldots, y_n]$ où chaque élément est calculé de la manière suivante :

$$y_i = \begin{cases} u_i \oplus v_i, & \text{si } i \leq n/2 \\ u_{i-n/2} \otimes v_{i-n/2}, & \text{sinon} \end{cases}$$

$$\text{où} \quad \begin{array}{l} u = dh(\oplus, \otimes, [x_1, \ldots, x_{n/2}]) \\ v = dh(\oplus, \otimes, [x_{n/2+1}, \ldots, x_n]) \end{array}$$

Le squelette **dh** implémente le fameux modèle d'échange de données appelé « *papillon* » (en anglais *butterfly*). Ce squelette demande un nombre important de communications entre les différents agents *MCAWorker* qui participent à son exécution. Par exemple, la figure 5.26 présente l'exécution du squelette *dh* sur un vecteur de huit éléments. Nous remarquons qu'il y a deux vecteurs intermédiaires avant d'obtenir le vecteur résultat contenant les valeurs $y_1, \ldots, y_n$. Chaque étape, permettant d'obtenir le vecteur suivant, demande quatre échanges de données suivant le modèle « *papillon* ». Un exemple d'échange est présenté par l'encadré de la figure 5.26. Les données sont échangées en utilisant deux opérateurs notés $\oplus$ et $\otimes$.

L'implantation de ce squelette est proposé par la méthode `dh` telle que

- `DVector<T> dh(DVector<T> input, BinOperator<T> oplus, BinOperator<T> otimes)`

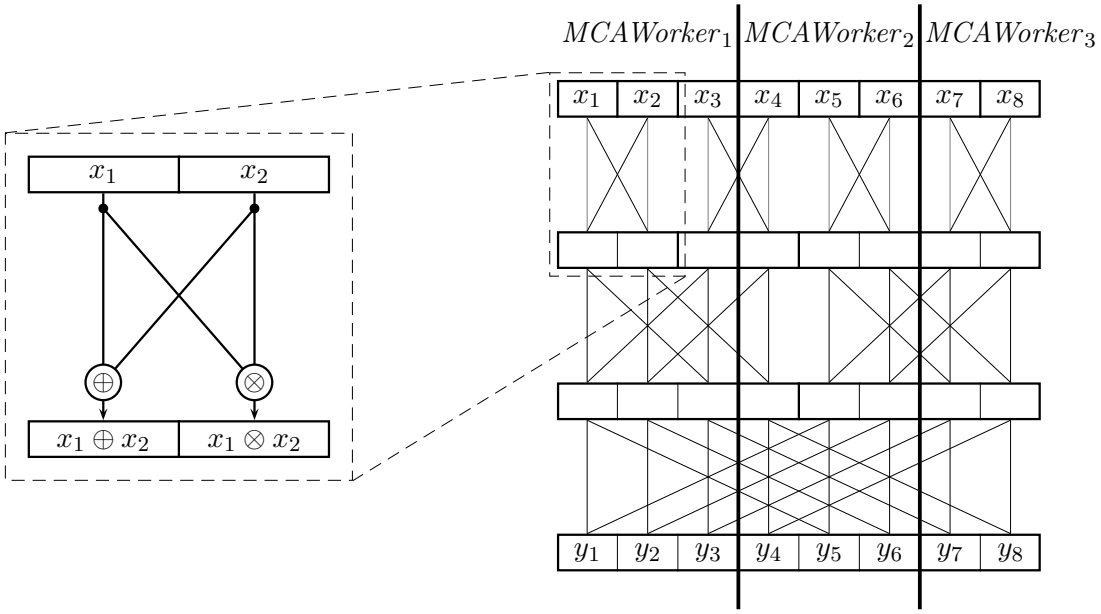


FIGURE 5.26 – Exécution du squelette *dh* sur un vecteur de huit éléments par trois agents *MCAWorker*.

Dans la section suivante nous utilisons l'implantation de ces différents squelettes pour réaliser une transformée de Fourier sur la plate-forme *MCA* installée sur la grille du LACL (cf. Section 5.1.1).

5.4.2 Un exemple d'application : Transformée de Fourier rapide

La transformée de Fourier est une opération qui associe un signal dans un domaine temporel et sa représentation dans un domaine fréquentiel. Différentes sortes de transformées de Fourier existent mais, pour notre étude de cas, nous nous intéressons uniquement à celles qui sont calculables. Notre choix s'est porté sur la *Transformée de Fourier Discrète* (*TFD*) [RG75] qui est une transformée de Fourier dans un domaine fini avec une fonction périodique. La *TFD* est un outil précieux qui est largement employé dans le traitement du signal numérique afin d'analyser les fréquences contenues dans les signaux. Cette solution est idéale pour le calcul de ses données par un ou plusieurs ordinateurs et donc adaptée à une grille de calcul. Pour calculer la *TFD* efficacement, nous utilisons la *transformée de Fourier rapide* (en anglais : *FFT* ou *Fast Fourier Transform*).

La FFT d'une liste $[x_0, \ldots, x_{n-1}]$ de taille $n = 2^m$ donne comme résultat une liste où le i-ème élément est défini comme le propose l'équation 5.8 [BG98]

$$(\mathrm{FFT}x)i = \sum_{k=0}^{n-1} x_k \omega_n^{ki} \tag{5.8}$$

où ω_n désigne la racine n-ième de l'unité [22] tel que $\omega_n = e^{\frac{2\pi i}{n}}$ (dans ce cas i est l'unité imaginaire, à ne pas confondre avec le i défini dans l'équation 5.8).

5.4.2.1 Définition de la *FFT* à l'aide de squelettes

La FFT peut être exprimée par une composition de squelettes [AG03] de la manière suivante

$$(\mathrm{FFT}x)_i = \begin{cases} (\mathrm{FFT}u)_i \oplus_{i,n} (\mathrm{FFT}v)_i & \text{si } i \leq n/2 \\ (\mathrm{FFT}u)_{i-n/2} \otimes_{i-n/2,n} (\mathrm{FFT}v)_{i-n/2} & \text{sinon} \end{cases} \tag{5.9}$$

où $u = [x_0, x_2, \ldots, x_{n-2}]$ et $v = [x_1, x_3, \ldots, x_{n-1}]$.

Cette formule est très proche de (eq. 2.39). Les opérateurs $\oplus$ et $\otimes$ sont ici paramétrés par i, position de l'élément dans la liste, et n, taille de la liste d'entrée. La définition du squelette *dh* (cf. Section 5.4.1.4) impose l'utilisation des mêmes opérateurs $\oplus$ et $\otimes$ pour tous les éléments de la liste. Si nous ne pouvons pas utiliser des opérateurs paramétrés avec le squelette *dh*, nous allons donc paramétrer les éléments de la liste. Nous commençons donc par transformer chaque élément x_i de la liste en un triplet (x_i, i, n) puis nous appliquons le squelette *dh* sur la liste des triplets. Nous définissons l'opérateur $\oplus$ appliqué sur deux triplets tel que

$$(x_1, i_1, n_1) \oplus (x_2, i_2, n_2) = (x_1 \oplus_{i_1,n_1} x_2, i_1, 2n_1) \text{ avec } x_1 \oplus_{i_1,n_1} x_2 = x_1 + \omega_{n_1}^{i_1} x_2$$

et l'opérateur $\otimes$ tel que

$$(x_1, i_1, n_1) \otimes (x_2, i_2, n_2) = (x_1 \otimes_{i_1,n_1} x_2, i_1, 2n_1) \text{ avec } x_1 \otimes_{i_1,n_1} x_2 = x_1 - \omega_{n_1}^{i_1} x_2.$$

La figure 5.27 montre comment le modèle d'échange « *butterfly* » est utilisé lors de la *FFT*.

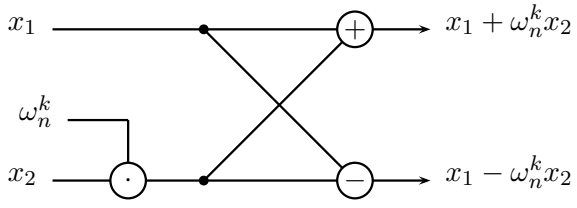


FIGURE 5.27 – Utilisation du modèle d'échange « butterfly » lors d'une *FFT*.

La FFT d'une liste l peut être exprimée de manière plus simple en une composition de squelettes telle que

$$FFT(l) = map(\pi_1) \circ dh(\oplus, \otimes) \circ mapidx(triple)\ (l) \tag{5.10}$$

22. En mathématiques, étant donné un nombre entier naturel non nul n, une racine n-ième de l'unité est un nombre complexe dont la puissance n-ième est égale à 1.

avec $\circ$ pour signifier une composition de fonctions de droite à gauche tel que $(f \circ g)(x) = f(g(x))$. La fonction *triple* transforme la valeur x_i en un triplet (x_i, i, n) et π_1 est la fonction de projection qui renvoie le premier élément d'un triplet.

Les deux opérateurs $\oplus$ et $\otimes$ (eq. 5.9), utilisent à chaque reprise la racine de l'unité ω_n^i. Afin d'éviter la répétition d'un même calcul, nous calculons à priori les valeurs de ω et les stockons dans une liste $\Omega = [\omega_n^1, \ldots, \omega_n^{n/2}]$ utilisée par chaque opérateur. La création de cette liste est définie par la composition de squelettes *repl* et *scan* telle que

$$\Omega = scan(*) \circ repl(\omega, n/2) \tag{5.11}$$

où $*$ une multiplication de nombres complexes. Nous pouvons alors écrire la composition suivante pour le calcul de la *FFT* :

$$FFT(l) = map(\pi_1) \circ dh(\oplus(\Omega), \otimes(\Omega)) \circ mapidx(triple)\ (l) \tag{5.12}$$

5.4.2.2 Implantation avec le framework *MCA*

Pour implanter la composition de squelettes permettant la résolution d'une *FFT* (eq. 5.12), nous avons défini la classe `FFTMasterAgent`, sous-classe de `SkeletonMasterAgent`, dont la méthode `execute` (cf. Figure 5.28) utilise les implantations des différents squelettes présentés dans la section 5.4.1.

```
1  DVector<Complex> input = computationCase.getData("input");
2  int size = input.getSize();
3  Complex rootUnity = new Complex(Math.cos(1), Math.sin(1));
4  DVector<Complex> r = repl(rootUnity, size / 2, new ComplexVectorFormat(), size / 2);
5  DVector<Complex> w = scan(r, new MultComplexOp());
6  DVector<Triple<Complex, Integer, Integer>> triples =
7      mapidx(input, new TripleFunction(size), new TripleVectorFormat());
8  BinOperator oplus = new OPlus(w);
9  BinOperator otimes = new OTimes(w);
10 DVector<Triple<Complex, Integer, Integer>> dh = dh(triples, oplus, otimes);
11 DVector<Complex> result = map(dh, new PiFunction(), new ComplexVectorFormat());
12 computationCase.finish();
13 return result;
```

FIGURE 5.28 – Code source de la méthode `execute` de la classe `FFTMasterAgent`.

Le vecteur `input` (l. 1 de la Figure 5.28), instance de `DVector`, contient les données sur lesquelles est réalisée la *FFT*. Dans un premier temps, il est nécessaire de calculer Ω (eq. 5.11) en utilisant les méthodes `repl` et `scan` (l. 3 à 5). Notons l'utilisation de la classe `Complex` fournie par la bibliothèque

Apache Commons Math [Com]. Nous avons développé deux classes héritant de classe `DataFormat` (cf. Figure 4.26) :

– la classe `ComplexVectorFormat` pour lire et écrire des nombres complexes ;
– la classe `TripleVectorFormat` pour lire et écrire des triplets contenant un nombre complexe et deux entiers.

La classe `MultComplexOp` (cf. Figure 5.29) implante l'opération binaire $*$ qui multiplie deux nombres complexes.

```
1 package org.mca.example.fft;
2
3 public class MultComplexOp implements BinOperator<Complex>{
4   public Complex execute(Complex a, Complex b) {
5     return a.multiply(b);
6   }
7 }
```

FIGURE 5.29 – Implantation de l'opération binaire $*$ (eq. 5.11).

La *FFT* est ensuite réalisée en trois étapes :

– Les données d'entrée sont transformées en une liste des triplets à l'aide de la méthode `mapidx`. La fonction *triple* (eq. 5.10) est implantée par la classe `TripleFunction` (cf. Figure 5.30a)
– Les classes `OPlus` et `OTimes` (l. 8 et 9) sont deux classes implantant l'interface `BinOperator` (cf. Figure 5.25) et représentent respectivement les opérations $\oplus(\Omega)$ et $\otimes(\Omega)$ utilisées lors de l'exécution du squelette *dh* (l. 10).
– Le solution de la *FFT* est la liste des valeurs obtenues par l'exécution du squelette *map* (via la méthode `map`). La fonction π_1 (eq. 5.10), qui retourne le premier élément d'un triplet, est implantée par la classe `PiFunction` (cf. Figure 5.30b).

Cet exemple d'application met en valeur l'utilisation de notre framework par une nouvelle étude de cas. La structure de notre framework est telle qu'elle conduit le concepteur dans une approche objet où il a été amené à décrire différentes classes (cf. Figure 5.25) :

– une classe pour le type de tâche du cas de calcul, ici la classe `SkeletonTask` ;
– autant de classe définissant un *ComputeAgent* qu'il y a de codes différents exécutés par les tâches du cas de calcul, ici les classes `SkeletonMasterAgent` et la classe `SkeletonAgent` et ses sous-classes.

La construction de cette étude de cas s'est effectuée en une semaine complète de travail. Nous espérons que cette simplicité d'approche amènera de nouveaux expérimentateurs à évaluer notre framework *MCA* pour de nouvelles études comme par exemple le calcul de Choleski [DM07], la résolution d'équation d'ordre 3 [DM08], etc ...

```
1  package org.mca.example.fft;
2
3  public class TripleFunction implements BinFunction<Integer,Complex,Triple<Complex,Integer,Integer>>{
4   private int n;
5   public TripleFunction(int n) {
6    this.n = n;
7   }
8   public Triple<Complex,Integer,Integer> execute( Integer index, Complex complex){
9    return new Triple<Complex,Integer,Integer>(complex,index,n);
10  }
11 }
```

(a) Fonction *triple*

```
1  package org.mca.example.fft;
2
3  public class PiFunction implements Function<Triple<Complex,Integer,Integer>,Complex>{
4   public Complex execute( Triple<Complex,Integer,Integer> triplet){
5    return triplet.getFirst();
6   }
7  }
```

(b) Fonction π_1

FIGURE 5.30 – Implantation des fonctions *triple* et π_1 (eq. 5.10).

5.5 Conclusion

L'évaluation d'un framework le calcul numérique est un travail multi-facette que nous avons abordé de manière constructive. Nous avons fait le choix de construire nos outils de mesure afin de perturber le moins possible les codes sources. Le développement de nouveaux aspects permet aisément de séparer les concepts. La possibilité d'injecter les greffons au chargement apporte l'adaptation de la prise de mesure à la taille du cas de calcul.

Nous avons fait le choix de nous limiter à trois des études de cas les plus significatives. La résolution de l'équation de Laplace a permis de quantifier le surcoût induit par l'adaptation de l'architecture pour un même cas de calcul. Nous avons constaté que ce surcoût pouvait être négligé dans une première approche. La gain lors de l'emploi d'un plus grand nombre de ressources de calcul est bien supérieur. Cet exemple montre l'intérêt de notre framework *MCA* pour la recherche d'une configuration optimale ou la comparaison de l'architecture de calcul.

L'approximation du calcul de π par une méthode de *Monte-Carlo* a mis en évidence la capacité de tolérance aux pannes. Cette propriété de transparence de panne est ainsi validée au cours des exécutions de cette étude. Nous observons que le calcul se poursuit avec des ressources en moins pendant l'exécution. De même, l'apport de nouvelles ressources est aussi pris en compte. Cette exemple montre l'aspect de notre framework pour des exécutions dans un cadre non fiable ou dans un contexte dit évolutif. C'est alors la transparence d'échelle qui est mis en œuvre.

Le développement de squelettes algorithmiques souligne la simplicité d'approche pour un concepteur au projet *MCA*. Faire des expérimentations est un exercice intéressant pour la validation, mais si l'utilisation n'est possible que par un expert du domaine alors, quelque soit la qualité du travail et des mesures, il passerait dans les oubliettes de la recherche. Nous avons fait un effort particulier pour que les évaluations puissent être faites par un informaticien aguerri aux technologies objets (ce qui n'est pas très limitatif de nos jours). Cette étude de cas permet de comptabiliser le temps de développement, la charge de travail que peut prendre une évaluation de notre framework en vue d'une application plus ambitieuse.

À défaut de transition immédiate entre le créateur de notre framework et ses utilisateurs, nous pensons que ces études participent à tisser des liens avec d'autres personnes des domaines du calcul numérique. Il est ainsi possible de monter des études comparatives, de participer au lancement de nouvelles études, mais aussi d'illustrer des concepts complexes de résolutions numériques par l'obtention de benchmarks spécifiques.

Conclusion et Perspectives

Notre travail de thèse aboutit à la définition d'une plate-forme logicielle pour la résolution de cas de calcul numérique dans un environnement distribué hétérogène telles que peuvent le proposer des grilles de calcul.

Notre étude du contexte de recherche dans lequel se situe cette thèse souligne que l'utilisation d'une grille de calcul donne à ses utilisateurs une grande puissance de calcul mais nécessite de mettre en place des applications distribuées capables d'exploiter les ressources disponibles. Si différents modèles de programmation existent pour définir l'architecture d'une application exécutée dans un environnement distribué, les besoins qui en ressortent peuvent se classer en terme de transparence.

La recherche des transparences les plus adaptées à l'utilisateur et au cas de calcul est un thème transverse au travail présenté dans cette thèse. Vouloir obtenir l'ensemble des transparences possibles reste un idéal, c'est pourquoi définir le meilleur compromis et le mettre en œuvre est notre objectif. Nos expériences passées, conjuguées aux travaux de recherche présentés ici, nous donnent matière à structurer les exigences des cas de calcul que nous traitons. Les transparences d'accès et de localisation offrent les aptitudes pour gérer de manière simple une grille de calcul. La transparence de mobilité apporte l'adaptabilité qui manque le plus souvent aux plates-formes existantes. Dans ce but, notre architecture logicielle type utilise des agents mobiles. Ce choix apporte une solution à la transparence de performance en exploitant au mieux les ressources de calcul disponibles dans une grille de calcul.

Contributions

Définition formelle d'une architecture logicielle

Nos spécifications formelles écrites en π-calcul polyadique d'ordre supérieur fournissent une **définition formelle d'une architecture logicielle pour la résolution de cas de calcul numérique**. Un système distribué bâti sur cette architecture est alors capable de résoudre différents cas de calcul numérique simultanément. En partant du modèle classique Maître-Travailleurs, nous décrivons une architecture capable de s'adapter aussi bien au nombre de ressources disponibles qu'au nombre de cas de calcul qu'elle doit traiter. Ainsi, la *transparence de localisation*,

la *transparence d'échelle* ou encore la *transparence d'accès* sont mises en valeur au travers de la spécification de termes π-calcul.

Définition d'une transformation d'une spécification en π-calcul d'ordre supérieur vers un réseau d'automates temporisés

La définition complète de la transformation d'une spécification en π-calcul d'ordre supérieur vers un réseau d'automates temporisés est une autre contribution de cette thèse. En effet, la définition de trois opérateurs de transformation et de leurs règles associées en fonction du contexte de transformation constituent un résultat offrant une aide à la preuve de propriétés temporelles. Ceci nous conduit à réaliser la transformation de notre spécification π-calcul dans le but de vérifier deux propriétés de transparence essentielles à notre projet :

- la *transparence de localisation* est une propriété non fonctionnelle. Elle autorise les utilisateurs à se détacher de contraintes matérielles telles que le placement de code sur un processus ou le découpage des données en fonction d'une architecture.
- la *transparence d'échelle* revêt un caractère essentiel dans la tolérance aux pannes. Nous constatons ainsi que notre système est apte à prendre en compte de nouvelles ressources de calcul et à les exploiter. De manière analogue notre système s'adapte à la perte de ressource de calcul en redistribuant les tâches et les données sur les ressources restantes.

Ces propriétés sont établies en utilisant notre réseau d'automates temporisés et avec l'emploi d'un outil reconnu dans le monde de la preuve par model-checking, l'outil UppAal. Ainsi, nous validons la possibilité d'établir des propriétés liées à la mobilité de code via un réseau d'automates.

Développement d'un framework Java

La modélisation formelle réalisée dans la première partie de cette thèse rend possible **le développement d'un framework proposant des outils en vue de la résolution de cas de calcul numérique dans un environnement distribué hétérogène**. L'architecture proposée par notre framework, nommé *MCA* (pour *M*obile *C*omputing *A*rchitecture), apporte les transparences décrites dans nos modèles auxquelles s'ajoutent de nouvelles propriétés de transparences provenant de nos choix de réalisation :

- Les composants *MCAWorker* apportent la *transparence de panne* et la *transparence d'échelle*. En effet, ceux-ci sont distribués de manière quelconque sur le réseau, leur nombre varie de manière dynamique sans nuire à la terminaison d'un cas de calcul. L'indisponibilité d'un agent de ce type est invisible à l'utilisateur.
- Les données d'un cas de calcul, mise en œuvre via l'interface *DataPart*, et de ses implantations locale et distante, offrent une *transparence d'accès* et une *transparence de localisation* car les agents *MCAWorker* ne connaissent pas l'emplacement où les données associées sont conservées. Les distributions des données et des tâches sont invisibles à l'utilisateur.

– Le composant *ComputeAgent* offre une *transparence de mobilité* car il permet à un agent *MCAWorker* d'exécuter un code sans que la migration n'ait été visible des autres *MCAWorker*. Ainsi, la possibilité de déplacer du code natif est une avancée considérable par rapport aux outils de simulation numériques existants et apporte l'adaptation au contexte d'exécution. L'utilisateur n'a pas le connaissance du lieu d'exécution du code de calcul.
– Enfin, l'utilisation des *spaces* comme élément central de notre architecture ajoute une *transparence de réplication de données* avec l'apport de la persistance de ses données et l'utilisation de transactions distribuées.

Développement d'outils de mesure

Pour réaliser l'évaluation de notre framework, **nos propres outils de mesure sont développés dans le but de réduire au maximum les effets des perturbations sur le code source des différents cas de calcul**. La séparation des concepts offerte par la programmation orientée aspect (*POA*) permet alors d'injecter nos métriques au chargement des composants. Cette technique apporte l'adaptation de la prise de mesure à la taille du cas de calcul.

Notre évaluation du framework *MCA* se lit au travers de trois cas d'étude :

– La résolution de l'équation de Laplace afin de quantifier le surcoût induit par l'adaptation de l'architecture pour un même cas de calcul. Notre constat est que le surcoût peut être négligé par rapport au gain apporté par l'emploi d'un plus grand nombre de ressources de calcul. Cet exemple montre l'intérêt de notre framework *MCA* pour la recherche d'une configuration optimale ou la comparaison de l'architecture de calcul.
– L'approximation du calcul de π par une méthode de *Monte-Carlo* met en évidence la capacité de tolérance aux pannes de notre famework. Nous observons qu'un calcul se poursuit avec des ressources de calcul en moins au cours de son exécution. De plus, l'apport de nouvelles ressources est aussi pris en compte.
– Enfin, le développement de squelettes algorithmiques fait ressortir la simplicité d'approche offerte à un concepteur voulant utiliser notre framework *MCA*. Cet étude de cas nous permet de comptabiliser le temps de développement et la charge de travail nécessaire au développement d'un nouveau cas de calcul dans le but d'effectuer sa résolution à l'aide de notre plate-forme.

Faire le point sur les apports d'un travail de plusieurs années n'a d'intérêt que dans l'étude des perspectives qu'il offre sur l'avenir. Celles-ci sont nombreuses car nos travaux passés sont multi-facettes au carrefour de la simulation numérique, de l'informatique distribuée et du génie logicielle.

Perspectives

Pour illustrer les perspectives de recherche de nos travaux, nous choisissons de présenter trois exemples.

Automatisation d'une méthode de modélisation

Notre expérience en modélisation de cas de calcul numérique nous incite à définir une approche structurée de modélisation. Notre première perspective est de mettre en place une démarche de modélisation pour faciliter la preuve de propriété réalisée à partir d'une spécification π-calcul. D'autre part, nous souhaitons piloter les transformations d'une spécification π-calcul d'ordre supérieur vers un réseau d'automates temporisés. Un assistant à la preuve de propriété temporelle dédié au calcul numérique est actuellement absent des outils de simulation (cf. Chapitre 1). Ces outils ne portent que sur la validation pour la construction de campagne de tests.

Étude de nouvelles propriétés

Une seconde perspective porte sur l'écriture et la preuve de nouvelles propriétés liée à la *transparence de mobilité* et ainsi montrer que les ressources telles que les données ou les codes de calcul peuvent se déplacer sans qu'un utilisateur en ait connaissance. Nos simplifications apportées à la spécification du chapitre 2 ont eu pour objectif de rendre lisible l'application de notre opérateur de transformation afin que les automates résultant soient de taille à être présentés dans cette thèse. En conservant la spécification telle qu'elle a été écrite au chapitre 2, nous pouvons dans une étude de propriété plus ambitieuse, mettre en valeur cette propriété de mobilité. L'utilisateur ne sera plus l'observateur conscient de la mobilité de données ou de code, mais l'utilisateur averti que des propriétés non fonctionnelles de transparence assure la gestion de son cas de calcul. Le résultat qu'il obtient en fin de simulation est le fruit de la gestion de l'ensemble des ressources sur une architecture partagée de calcul.

Portage d'un code existant

Une troisième perspective est basée sur l'utilisation de code natif. Trop de code numérique ont été écrit il y a longtemps. Il n'est pas envisageable de les ré-écrire dans un nouveau langage car leur écriture est satisfaisante. Nous souhaitons établir que ces mêmes codes numériques peuvent être utilisés dans un contexte tolérant aux fautes. Aussi notre perspective est de prendre un code de calcul plus ambitieux tel qu'un calcul de *FDTD* avec ses valeurs de benchmark et d'en faire une évaluation comparative. Notre but est de montrer que tout code (par exemple en Fortran) peut être mis en situation d'être exécuté par un ensemble d'agents mobiles. L'exploitation la plus satisfaisante des ressources de calcul aboutit à des mesures qui peuvent alors être comparées aux valeurs de référence.

Les perspectives ont l'intérêt de proposer des pistes d'avenir à un travail existant. Bien entendu, il ne faut pas omettre les rencontres, les partenariats, voire les futures affectations qui transforment le prévisible en inattendu. L'essentiel reste que cette recherche soit vivante et se poursuive par mes soins et tous ceux souhaiteront en profiter.

Bibliographie

[ABB+01] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise et W. Yi. UPPAAL - Now, Next, and Future. *In Modeling and Verification of Parallel Processes*, volume 2067 de *Lecture Notes in Computer Science*. Springer, 2001.

[ACCK01] J. Algesheimer, C. Cachin, J. Camenisch et G. Karjoth. Cryptographic security for mobile code. *In Proceedings of the 2001 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2001.

[AD94] R. Alur et D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183 – 235, 1994.

[Aff12] M. Affouf. *Scilab by Example*. CreateSpace Independent Publishing Platform, 2012.

[AG03] M. Alt et S. Gorlatch. Using skeletons in a java-based grid system. *In Proceedings of 9th International Euro-Par Conference*, volume 2790 de *Lecture Notes in Computer Science*, pages 742–749. Springer, 2003.

[All07] G. Allaire. *Analyse numérique et optimisation : Une introduction à la modélisation mathématique et à la simulation numérique*. Ecole Polytechnique, 2007.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell et C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[AN01] A. Arnold et D. Niwinski. *Rudiments of mu-calculus*. North Holland, 2001.

[BBC+06] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel et R. Quilici. Programming, Composing, Deploying for the Grid. *In Grid Computing : Software Environments and Tools*, pages 205 – 229. Springer, 2006.

[BBC07] L. Baduel, F. Baude et D. Caromel. Asynchronous Typed Object Groups for Grid Programming. *International Journal of Parallel Programming*, 35(6):573–614, décembre 2007.

[BBD+02] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, et W. Yi. UPPAAL Implementation Secrets. *In Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 3 – 22, 2002.

[BC02] E. Bierman et E. Cloete. Classification of malicious host threats in mobile agent computing. *In Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, SAICSIT '02, Republic of South Africa, 2002.

[BCF+05] P. Bellavista, A. Corradi, C. Federici, R. Montanari et D. Tibaldi. Security for mobile agents : Issues and challenges. *In* M. Ilyas et I. Mahgoub, éditeurs. *Mobile Computing Handbook*, chapitre 39, pages 941–960. CRC Press, 2005.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis et S. Yovine. KRONOS : A model-checking tool for real-time systems. *In CAV'98 : Proceedings of the 10th International Conference on Computer Aided Verification*, 1998.

[Ber09] M. Bernichi. *Surveillance logicielle à base d'une communauté d'agents mobiles*. Thèse de doctorat, Université Paris XII, 2009.

[BG98] H. Bischof et S. Gorlatch. A Generic MPI Implementation for a Data-Parallel Skeleton : Formal Derivation and Application to FFT. *Parallel Processing Letters*, 8(4):447–458, 1998.

[BGK+96] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson et W. Yi. Verification of an audio protocol with bus collision using UPPAAL. *In CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification*, pages 244–256. Springer-Verlag, 1996.

[BI02] G. Bernard et L. Ismail. Apport des agents mobiles à l'exécution répartie. *Technique et Science Informatiques*, pages 771–796, 2002.

[Bli] Blitz JavaSpaces. http ://www.dancres.org/blitz/.

[BM02] A. Barbu et F. Mourlin. Mobile properties and temporal logic. *In EurAsia ICT 2002, Workshop on Language, Software and Data Engineering*. Austrian Computer Society, 2002.

[BMW98] I. Biehl, B. Meyer et S. Wetzel. Ensuring the integrity of agent-based computations by short proofs. *In MA '98 : Proceedings of the Second International Workshop on Mobile Agents*, volume 1477 de *Lecture Notes in Computer Science*, pages 183–194, Stuttgart, Germany, 1998. Springer-Verlag.

[BR05] P. Braun et W. R. Rossak. *Mobile Agents : Basic Concepts, Mobility Models, and the Tracey Toolkit*. Morgan Kaufmann Publishers In, 2005.

[Bso] Spécification BSON. http ://bsonspec.org/.

[BVBF13] A. Boudjadar, F. Vaandrager, J.-P. Bodeveix et M. Filali. Extending UPPAAL for the Modeling and Verification of Dynamic Real-Time Systems. *In* F. Arbab et M. Sirjani, éditeurs. *Fundamentals of Software Engineering*, Lecture Notes in Computer Science, pages 111–132. Springer Berlin Heidelberg, 2013.

[CGH+98] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris et G. Tsudik. Itinerant agents for mobile computing. *In Readings in agents*, pages 267–282. Morgan Kaufmann Publishers Inc., 1998.

[Col89] M. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press, 1989.

[Com] The Apache Commons Mathematics Library. http ://commons.apache.org/proper/commons-math/.

[Dav06] G. David. *Modélisation dynamique des modèles physiques et numériques pour la simulation en électromagnétisme. Application dans un environnement de simulation intégrée : SALOME*. Thèse de doctorat, INP Grenoble, novembre 2006.

[DBLY02] A. David, G. Behrmann, K. G. Larsen et W. Yi. New Uppaal Architecture. *In Proceedings of the Workshop on Real-Time Tools*. Uppsala University Technical Report Series, 2002.

[DBLY03] A. David, G. Behrmann, K. G. Larsen et W. Yi. A Tool Architecture for the Next Generation of Uppaal. *In 10th Anniversary Colloquium. Formal Methods at the Cross Roads : From Panacea to Foundational Support*, LNCS, pages 352–366, 2003.

[DC10] M. Dirolf et K. Chodorow. *MongoDB : The Definitive Guide*. O'Reilly Media, 2010.

[DM07] C. Dumont et F. Mourlin. A Mobile Computing Architecture for Numerical Simulation. *In UBICOMM '07 : Proceedings of the International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 68–74, November 2007.

[DM08] C. Dumont et F. Mourlin. Space Based Architecture for Numerical Solving. *In CIMCA 2008 : Proceedings of the International Conference on Computational Intelligence for Modelling Control and Automation*, pages 309–314, Vienna, Austria, December 2008. IEEE Computer Society.

[DPHBB12] N. De Palma, D. Hagimont, F. Boyer et L. Broto. Self-Protection in a Clustered Distributed System. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2): 330–336, Feb 2012.

[EG02] F. B. Engelhardtsen et T. Gagnes. Using JavaSpaces to create adaptive distributed systems. *In NIK'2002 : Norsk Informatikkonferanse*, 2002.

[EH86] E. A. Emerson et J. Y. Halpern. « Sometimes » and « Not Never » Revisited : On Branching versus Linear Time Temporal Logic. *Journal of the Association for Computing Machinery*, 33:151–178, 1986.

[EHRLR80] L. D. Erman, F. Hayes-Roth, V. R. Lesser et D. R. Reddy. The Hearsay-II speech-understanding system : Integrating knowledge to resolve uncertainty. *ACM Comput. Surv.*, 12(2):213–253, 1980.

[Esn05] A. Esnard. *Analyse, conception et réalisation d'un environnement pour le pilotage et la visualisation en ligne de simulations numériques parallèles.* Thèse de doctorat, Université de Bordeaux I, décembre 2005.

[Fer95] J. Ferber. *Les systèmes multi-agents. Vers une intelligence collective.* InterEditions, Paris, 1995.

[FGMM03] G. Ferrari, S. Gnesi, U. Montanari et M.Pistore. A model checking verification environment for mobile processes. *ACM Transactions on Software Engineering and Methodology*, 12(4):440–473, 2003.

[FGS96] W. M. Farmer, J. D. Guttman et V. Swarup. Security for mobile agents : Authentication and state appraisal. *In Proceedings of the 4th European Symposium on Research in Computer Security : Computer Security*, ESORICS '96, pages 118–130, London, UK, 1996. Springer-Verlag.

[FHA99] E. Freeman, S. Hupfer et K. Arnold. *JavaSpaces™ Principles, Patterns, and Practice.* Prentice Hall, 1999.

[FK98] I. Foster et C. Kesselman. *The Grid : Blueprint for a new Computing Infrastructure.* Morgan Kaufman Publishers, 1998.

[FKK11] A. Freier, P. Karlton et P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), 2011.

[FPV98] A. Fuggetta, G. P. Picco et G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[GD10] K. Gama et D. Donsez. A Self-healing Component Sandbox for Untrustworthy Third Party Code Execution. *In* L. Grunske, R. Reussner et F. Plasil, éditeurs. *Component-Based Software Engineering*, volume 6092 de *Lecture Notes in Computer Science*, pages 130–149. Springer Berlin Heidelberg, 2010.

[Gel93] D. Gelernter. *Mirror Worlds.* Oxford University Press, 1993.

[Gig] GigaSpaces Platform. http ://www.gigaspaces.com/.

[Gig11] Inside GigaSpaces XAP Technical Overview and Value Proposition. Rapport technique, GigaSpaces White Paper, 2011.

[GLT99] W. Gropp, E. L. Lusk et R. Thakur. *Using MPI-2 : Advanced Features of the Message Passing Interface.* The MIT Press, 1999.

[HHWT95] T. A. Henzinger, P.-H. Ho et H. Wong-Toi. A user guide to HYTECH. *In TACAS '95 : Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 41–71. Springer-Verlag, 1995.

[HNS+94] T. A. Henzinger, X. Nicollin, J. Sifakis, et S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[Hoa86] C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1986.

[HSLL97] K. Havelund, A. Skou, K. G. Larsen et K. Lund. Formal Modeling and Analysis of an Audio/Video Protocol : An Industrial Case Study Using Uppaal. *In Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Society Press, 1997.

[IBM06] IBM. An Architectural Blueprint for Autonomic Computing, juin 2006.

[ISO09] ISO/IEC 10746-3 :2009. Information technology – Open distributed processing – Reference model : Architecture, 2009.

[JET] Java™ Execution Time Measurement Library. http ://jetm.void.fm/.

[JP01] J. Jaworski et P. Perrone. *Java™ Security*. CampusPress, 2001.

[KC03] J. O. Kephart et D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[LAFH04] H. Lee, J. Alves-Foss et S. Harrison. The Use of Encrypted Functions for Mobile Agent Security. *In Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, 2004.

[Lat11] C. Lattner. *The Architecture of Open Source Applications*, chapitre 11, pages 155–170. Amy Brown and Greg Wilson, 2011.

[LBCC08] L. Liquori, D. Borsetti, C. Casetti et C.-F. Chiasserini. An Overlay Architecture for Vehicular Networks. *In* A. Das, H. Pung, F. Lee et L. Wong, éditeurs. *NETWORKING 2008 Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet*, volume 4982 de *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin Heidelberg, 2008.

[LD98] H. Luzet et L. M. Delves. Porting the SEMC3D Electromagnetics Code to HPF. *In Euro-Par*, Lecture Notes in Computer Science, pages 1140–1148. Springer, 1998.

[Leg56] J. Legras. *Techniques de résolution des équations aux dérivées partielles*. Paris, Dunod, 1956.

[LGH12] C. Li, F. Gava et G. Hains. Implementation of Data-Parallel Skeletons : A Case Study Using a Coarse-Grained Hierarchical Model. *In Proceedings of the 2012 11th International Symposium on Parallel and Distributed Computing*, ISPDC '12, pages 26–33. IEEE Computer Society, 2012.

[Lia99] S. Liang. *Java Native Interface : Programmer's Guide and Specification*. Prentice Hall, 1999.

[LMR00] S. Loureiro, R. Molva et Y. Roudier. Mobile Code Security. *In ISYPAR 2000 (4ème Ecole d'Informatique des Systèmes Parallèles et Répartis)*, Toulouse, 2000.

[LP97] H. Lönn et P. Pettersson. Formal Verification of a TDMA Protocol Start-Up Mechanism. *In PRFTS'97 : Proceedings of Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242. IEEE Computer Society, 1997.

[LPY97] K. G. Larsen, P. Pettersson et W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[LPY98] M. Lindahl, P. Pettersson et W. Yi. Formal design and analysis of a gear controller. *In TACAS'98 : Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 281–297. Springer-Verlag, 1998.

[Mag07] M. Magnin. *Réseaux de Petri à chronomètres - Temps dense et temps discret*. Thèse de doctorat, École Centrale de Nantes, 2007.

[Mai13] C. M. Mair. *Equations différentielles avec transformées de Laplace : Théorie des solutions*. Presses Academiques Francophones, 2013.

[Mat98] R. Mateescu. *Vérification des propriétés temporelles*. Thèse de doctorat, Institut National Polytechnique de Grenoble, 1998.

[MB02] J. Mir et J. Borrell. Protecting general flexible itineraries of mobile agents. *In Proceedings of the 4th International Conference Seoul on Information Security and Cryptology*, ICISC '01, pages 382–396, London, UK, 2002. Springer-Verlag.

[Mic88] S. Microsystems. RPC : Remote Procedure Call Protocol specification : Version 2. RFC 1057 (Informational), juin 1988.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil91] R. Milner. The polyadic pi-calculus : a tutorial. Rapport technique, Logic and Algebra of Specification, 1991.

[Mil99] R. Milner. *Communication and Mobile Systems : the pi-Calculus*. Cambridge University Press, 1999.

[Mis06] S. Misljencevic. *Jini Service Container*. Thèse de doctorat, Universiteit Antwerpen, 2006.

[MPW92] R. Milner, J. Parrow et D. Walker. A calculus of mobile processes, parts i and ii. *Journal of Information and Computation*, 100:1–77, 1992.

[New06] J. Newmarch. *Foundations of Jini™ 2 Programming*. Apress, 2006.

[NL98] G. C. Necula et P. Lee. Safe, Untrusted Agents Using Proof-Carrying Code. *In Mobile Agents and Security*, pages 61–91, London, UK, 1998. Springer-Verlag.

[Obj04] Object Management Group (OMG). CORBA/IIOP Specification v3.0.3. Rapport technique formal/2004-03-01, Object Management Group, Mars 2004.

[Pac97] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997.

[PH06] M. Parashar et S. Hariri. *Autonomic Computing : Concepts, Infrastructure, and Applications*. CRC Press Inc, 2006.

[PP06] F. Peschanski et D. Poitrenaud. *Vérification de systèmes infinis*, chapitre 9, pages 213–250. Hermès Lavoisier, 2006.

[PRS04] R. Pawlak, J.-P. Retaillé et L. Seinturier. *Programmation orientée aspect pour Java / J2EE*. Eyrolles, 2004.

[RG75] L. R. Rabiner et B. Gold. *Theory and Application of Digital Signal Processing*. Prentice Hall, 1975.

[RV00] H. Reiser et G. Vogt. Security requirements for management systems using mobile agents. *In Proceedings of the Fifth IEEE Symposium on Computers and Communications*, ISCC '00, Antibes, France, 2000. IEEE Computer Society.

[San92] D. Sangiorgi. From π-calculus to Higher-order π-calculus — and back. *In Proc. TAPSOFT '93*, numéro 668 de LNCS. Springer – Verlag, 1992.

[Sch00] P. Schnoebelen. *Vérification de logiciels : Techniques et outils du model-checking*. Vuibert, 2000.

[ST98] T. Sander et C. F. Tschudin. Protecting mobile agents against malicious hosts. *In Mobile Agents and Security*, pages 44–60. Springer-Verlag, 1998.

[Tiw11] S. Tiwari. *Professional NoSQL*. John Wiley & Sons Ltd, 2011.

[Whi99] J. E. White. Telescript technology : mobile agents. *In Mobility*, pages 460–493. ACM Press/Addison-Wesley Publishing Co., 1999.

[Whi10] T. White. *Hadoop : The Definitive Guide*. O'Reilly Media, 2010.

[WT00] J. Waldo et The Jini Team. *The Jini™ Specifications*. Addison-Wesley Professional, 2nd édition, 2000.

[Zac03] J. Zachary. Protecting Mobile Code in the Wild. *IEEE Internet Computing*, 7(2):78–82, March 2003.

Résumé

Ce travail appartient au domaine de la simulation numérique sur des plates-formes d'exécution distribuées hétérogènes telles que des grilles de calcul. Ce type de plate-forme se caractérise par des possibles changements de condition d'exécution et par une probabilité importante de défaillance de certains composants. Une application qui s'exécute dans un tel environnement se doit d'être adaptable à son contexte d'exécution et tolérante aux pannes.

Face à la complexité croissante de la mise en place de cas de calcul sur des grilles de calcul, nous proposons une plate-forme logicielle pour la résolution de cas de calcul numérique dans un environnement distribué hétérogène. Nos travaux apportent une solution qui se base sur un système d'agents mobiles, ce qui permet à une application de s'adapter au changement de son environnement d'exécution.

Dans un premier temps, nous utilisons le langage π-calcul d'ordre supérieur pour spécifier une « ferme de travailleurs » capable de participer à la résolution de tout type de cas de calcul.

Ensuite, nous énonçons des propriétés qui caractérisent le bon fonctionnement de ce système avec une logique temporelle TCTL. Pour cela, nous souhaitons modéliser notre système à l'aide d'automates temporisés à partir des termes définis par la spécification formelle en π-calcul. Dans ce but, nous définissons une transformation de termes écrits en π-calcul en automates temporisés. Les propriétés sont alors vérifiées avec l'outil UppAal.

Pour valider ce travail de modélisation, nous avons réalisé le framework *MCA* (pour *Mobile Computing Architecture*). Celui-ci propose un ensemble d'outils facilitant la mise en place de composants sur un environnement distribué hétérogène dans le but d'effectuer la résolution de cas de calcul. La librairie avec laquelle sont développés ces composants, qu'ils soient mobiles ou non, est implantée en *Java* et se base les technologies Jini et JavaSpaces.

Enfin, nous réalisons l'évaluation du framework *MCA* en procédant à la résolution de trois cas de calcul différents. Chacune de ces expériences, réalisées sur une grappe de 20 nœuds, nous permet de montrer les caractéristiques essentielles de notre framework : une simplicité de programmation, un faible surcoût en temps d'exécution sans l'activation de la tolérance aux pannes et une tolérance aux pannes efficace.

Mots-clés: Calcul parallèle, Grille de calcul, Agents mobiles, Tolérance aux fautes, Model checking, Spécifications formelles

Abstract

This work belongs to the domain of numerical simulation on heterogeneous distributed platforms such as grids. This type of platform is characterized by possible changes in execution conditions and a significant probability of some components failure. An application running in such an environment must be adaptable to its execution context and fault-tolerant.

Facing the growing complexity of implementing computation cases on grid computing, we propose a software platform which solves numerical computation cases in a distributed heterogeneous environment. Our work provides a solution based on a mobile agent system, which allows an application to adapt to change in its execution environment.

At first, we use the higher-order π-calculus language to specify a « farm of workers » able to take part in solving any type of computation case.

Then we set the properties that characterize the system's correct execution with a temporal logic TCTL. In order to do this, we perform a temporal modeling system based on terms defined by the formal specification in π-calculus. To achieve this transformation, we define a translation of terms written in π-calculus into timed automata. The properties are verified with the UppAal tool.

To validate this modeling work, we develop the *MCA* (for *Mobile Computing Architecture*) framework. It offers a set of tools which facilitate the implementation of distributed heterogeneous components in order to solve computation cases. These components, mobile or not, are developed with a library written in *Java* and which uses Jini and JavaSpaces technologies.

Finally, our framework is evaluated through the resolution of three different computation cases. Each of these experiments, performed on a 20 node cluster allow us to highlight our framework's main characteristics : programming simplicity, low overhead in execution time without the fault tolerance activation and efficient fault tolerance.

Keywords: Parallel computing, Grid computing, Mobile agents, Fault tolerance, Model checking, Formal specifications