JUXMEM : un service de partage transparent de données pour grilles de calcul fondé sur une approche pair-à-pair

Mathieu Jan

Numéro d'ordre de la thèse : 3453

THÈSE

présentée devant

L'UNIVERSITÉ DE RENNES 1

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention INFORMATIQUE

PAR

Mathieu Jan

Équipe d'accueil : projet PARIS, IRISA, Rennes

École Doctorale MATISSE

Composante universitaire : IFSIC/IRISA

JUXMEM : un service de partage transparent de données pour grilles de calcul fondé sur une approche pair-à-pair

SOUTENUE LE 20 novembre 2006 devant la commission d'Examen

COMPOSITION DU JURY

Monsieur Gabriel ANTONIU, Chargé de recherche, INRIA	directeur de thèse
Monsieur Luc BOUGÉ, Professeur, ENS Cachan/Antenne de Bretagne	directeur de thèse
Monsieur Franck CAPPELLO, Directeur de recherche, INRIA	rapporteur
Monsieur Raymond NAMYST, Professeur, Université de Bordeaux	rapporteur
Monsieur Thierry PRIOL, Directeur de recherche, INRIA	examinateur
Monsieur Bernard TRAVERSAT, Senior engineering, Sun Microsystems	examinateur

Remerciements

Voilà ! Cette première page achève la rédaction de mon manuscrit de thèse. Elle vise à remercier les personnes qui ont de près ou de loin permis sa réalisation.

Mes premiers remerciements vont tout d'abord aux membres du jury pour leur évaluation de mon travail. Je remercie Franck et Raymond d'avoir pris le temps pour lire ce manuscrit et faire les remarques indispensables, toujours constructives. Je remercie Bernard d'avoir fait un aussi long voyage depuis la Californie ainsi que pour son accueil plus que chaleureux lors de mon séjour dans l'équipe JXTA de Sun Microsystems. Je tiens également à remercier Thierry de m'avoir accepté dans sa dynamique équipe. Enfin, un grand merci à mes deux directeurs de thèse d'avoir cru en mes capacités pour la réalisation de ce travail. Merci Luc pour l'ensemble des remarques qui m'ont donné le goût travail bien fait ! Merci à toi Gabriel pour ton sérieux au travail. Tu m'as fait découvrir et apprécier le travail passionnant de chercheur.

Merci également à toutes les personnes que j'ai pu cotoyer à l'IRISA : une ambiance de travail plus qu'agréable a toujours été au rendez-vous. Tout d'abord un grand merci à toi Sébastien (Monnet) d'avoir été et d'être plus qu'un collègue de bureau. Ce travail n'aurait pas été possible sans toi. Merci Loïc, le jeunot :-), pour les moments de détente au cours des différentes missions. Courage ! Merci Christian pour tes questions pertinentes sur JUXMEM, mais aussi pour nos discussions relaxantes aux pauses. Je note que tu n'as pas réussi à me noyer :-). Je tiens à te remercier Hinde pour tes encouragements et je te les retourne. Merci Landry pour tous tes efforts sur JUXMEM. Merci également Sébastien (Lacour) pour tes remerciements : oui il y a bien eu retour sur investissement ! Enfin, merci à toute l'équipe d'administration du site rennais de Grid'5000 : David, Guillaume, Pacal et Yvon. Sans vous ce travail aurait été dépourvu de résultats pratiques.

Merci également aux personnes avec lesquelles j'ai pu travailler à l'extérieur de l'IRISA. Merci plus particulièrement à Eddy, mais aussi Frédéric, Holly, Raphaël et de manière plus générale à toute l'équipe DIET pour votre aide concernant la validation de JUXMEM. Merci à Mohamed, Henry, Mike, Rick et James pour le partage de leurs connaissances sur JXTA et leur accueil lors de mon séjour en Californie : *I wish it could be lunch time!* Je tiens également à te remercier Alexandre pour ton extrême disponibilité lors de mes utilisations de PadicoTM. Merci aux différentes personnes (plus particulièrement Pierre !) que j'ai pu rencontrer dans des réunions de travail ou des conférences. Vos remarques pertinentes m'ont permis de progresser. Enfin, merci à toutes les personnes qui ont contribué à la mise en place de la grille expérimentale Grid'5000 et merci à Grid'5000 pour son iPod nano ! :-)

Je tiens plus particulièrement à te remercier Hélène pour ton soutien et tes encouragements à chaque fois que j'ai eu des doutes sur mes choix professionnels. Je suis persuadé que les choix que nous avons faits sont les bons et nous mèneront là où nous voulons ! Merci papa et maman pour m'avoir encouragé à poursuivre mes études à travers un doctorat. Merci également papa, pour tes relectures plus qu'attentives de ce manuscrit, tu l'auras lu et relu plus que d'autres ! Merci à Claudine, Roger-Charles et Maxence pour les moments de détente familiale entre mes heures de rédaction chez vous.

Que les autres personnes ne se vexent pas elles méritent d'autres remerciements, mais il n'y a heureusement pas que le travail dans la vie !

Table des matières

Liste des définitions

Chapitre 1

Introduction

COMPRENDRE le fonctionnement du monde qui l'entoure est le premier moteur d'évolution de l'Homme ainsi que de ses découvertes. La complexité des phénomènes et des mécanismes à analyser l'a poussé à raffiner les modèles qu'il a établis. L'apparition de l'informatique a eu comme conséquence d'augmenter significativement ses capacités de modélisations, en permettant l'exécution plus rapide et plus précise de simulations des différents modèles proposés.

En 1965, Gordon Moore, cofondateur d'Intel, énonce dans une loi qui gardera son nom, le fait que le « *nombre de transistors des processeurs sur une puce de silicium double tous les deux ans* » [128]. Cette augmentation exponentielle des capacités de calcul s'est révélée exacte, et même sur une période de 18 mois et non 24. Par ailleurs, d'après la loi de Kryder[1], la capacité de stockage des machines double tous les 23 mois.

Parallèlement à cette course sans fin vers une puissance accrue des machines en calcul et stockage, et dans une dimension orthogonale, les réseaux de communications se sont développés. Ceci a permis la naissance du calcul distribué sur un ensemble de machines, première étape vers l'utilisation massive de la puissance combinée de celles-ci. Ce que l'on nomme les *grilles de calcul* en constitue l'étape suivante, mais à une plus grande échelle. Ces infrastructures permettent théoriquement d'offrir des capacités quasiment infinies de calcul et de stockage, par rapport aux besoins des applications scientifiques.

Toutefois, les grilles de calcul sont utilisables en pratique uniquement si les utilisateurs n'ont pas à se soucier du fonctionnement interne de ces infrastructures. En effet, cette puissance n'est en pratique ni infinie, ni localisée dans un même endroit géographique. Elle est distribuée, c'est-à-dire répartie sur un grand nombre de machines, qui sont hétérogènes et sujettes aux pannes. Pour les concepteurs d'applications, cela constitue un ensemble de contraintes fortes que doivent abstraire les modèles de programmation, sous peine d'être un frein au développement d'applications utilisatrices des grilles de calcul.

[1] De Mark Kryder, ingénieur chez la compagnie de disques durs Seagate.

Malgré les efforts menés pour développer des modèles de programmation adaptés, une de leurs principales limites reste *la gestion des données* utilisées par différents processus applicatifs répartis sur plusieurs machines. Cette limite est amplifiée lorsque les données sont partagées. En effet, outre le fait que le programmeur doit localiser la ou les machines qui les hébergent, indiquer les machines de destination pour les transférer, etc., il doit également maintenir la cohérence de l'ensemble des copies d'une donnée qui peuvent être accédées de manière concurrente. Ceci implique un ensemble de tâches de programmation particulièrement fastidieuses et de bas niveau comparées aux abstractions que fournissent les modèles de programmation pour la gestion des calculs.

Ainsi, cette illusion de puissance informatique infinie que fournissent les grilles de calcul, ne peut pleinement s'exprimer que par l'intégration aux modèles de programmation pour grilles d'un modèle adéquat de gestion des données.

Objectifs

Nous proposons donc d'intégrer aux modèles de programmation utilisés sur les grilles de calcul un *modèle d'accès transparent aux données* en proposant le concept de *service de partage de données*. L'objectif d'un tel service est d'abstraire les problèmes liés à la gestion des données sur les grilles de calcul et ainsi de faciliter la programmation des applications scientifiques distribuées. Le mot-clé de ce nouveau type de système est la *transparence* : transparence de localisation des données stockées dans une grille de calcul, transparence de la tolérance à la volatilité, transparence de la gestion de la cohérence. La conception d'un tel service a été initié dans le cadre du projet *Grid Data Service* (GDS [208]) de l'ACI Masse de Données, qui a débuté en 2003.

Ce travail vise plus particulièrement à définir notre approche pour la conception d'un service de partage de données, d'en proposer une mise en œuvre expérimentale et de l'évaluer. Notre approche du concept de service de partage de données s'inspire à la fois des systèmes pair-à-pair (P2P) et des systèmes à mémoire virtuellement partagée (MVP). Il s'agit par cette approche hybride de faire passer à l'échelle les propriétés intéressantes des systèmes à MVP (transparence d'accès, modèles et protocoles de cohérence), grâce à une organisation fondée sur une approche P2P. Afin de valider notre concept, nous proposons une architecture appelée JUXMEM (pour *Juxtaposed Memory*), ainsi qu'une implémentation s'appuyant sur la spécification P2P JXTA (pour *JuXTAposed*). Cette mise en œuvre a été validée par son intégration dans deux modèles de programmation, utilisés pour concevoir les applications s'exécutant sur les grilles de calcul : le modèle Grid-RPC et les modèles à base de composants. Le service de partage de données JUXMEM a été rendu public [218] sous la licence GNU LGPL et enregistré à l'Agence pour la Protection des Programmes (APP).

Par ailleurs, nous décrivons également dans ce manuscrit deux contributions annexes à cette contribution principale que représente JUXMEM. Ces contributions se placent dans le contexte d'utilisation de techniques P2P pour la construction d'intergiciels pour les grilles de calcul. Elles portent d'une part sur l'utilisation de la plate-forme P2P JXTA sur les grilles de calcul et d'autre part sur son évaluation et son optimisation dans ce contexte particulier d'exécution. L'objectif de ces contributions est d'adapter aux grilles de calcul le support utilisé pour la conception et la mise en œuvre de JUXMEM, afin d'optimiser les performances de ce dernier.

Contributions

Nous énumérons ici les différentes contributions de ce travail de doctorat.

Concept de service de partage de données pour grilles de calcul.
La gestion des données sur les grilles de calcul nécessite la conception de systèmes adaptés. En effet, l'architecture physique de ces environnements est bien différente de celle des grappes de machines, des supercalculateurs ou d'Internet par exemple. Par ailleurs, cette gestion doit être transparente pour les applications et cela sur plusieurs aspects, à savoir la localisation, la cohérence, la tolérance à la volatilité, la persistance, l'hétérogénéité, etc. Notre réponse à ce besoin consiste en la proposition du concept de service de partage de données pour grilles de calcul [2, 9, 10].

JUXMEM : une proposition d'architecture fondée sur une approche P2P.
Notre concept de service de partage de données s'accompagne d'une proposition d'architecture pour son implémentation. Celle-ci se fonde sur une approche P2P et utilise la bibliothèque P2P JXTA pour sa mise en œuvre. Cette implémentation a été intégrée et évaluée au sein de deux modèles de programmation utilisés pour concevoir des applications s'exécutant sur les grilles de calcul : le modèle Grid-RPC [1] et les modèles à base de composants [3, 4]. La première validation a été réalisée en collaboration avec Eddy Caron[2]. La seconde validation a quant à elle été réalisée en collaboration avec Christian Pérez[3].

Évaluation et améliorations des performances de la plate-forme JXTA.
Cette contribution est constituée de deux axes. Premièrement, l'utilisation des systèmes P2P pour concevoir des intergiciels pour les grilles de calcul est naturelle. En effet, les mécanismes P2P apportent flexibilité, passage à l'échelle et auto-organisation à ces intergiciels. Toutefois, l'utilisation de bibliothèques P2P sur les grilles de calcul nécessite leur adaptation aux caractéristiques physiques de ces environnements d'exécution. Notre contribution dans ce domaine consiste en une évaluation et une adaptation des couches de communication de la plate-forme P2P JXTA aux réseaux haute performance disponibles sur les grilles de calcul [6, 5]. Ces évaluations ont été initiées en collaboration avec David A. Noblet[4]. Le second axe de cette contribution concerne l'étude grande échelle de JXTA, menée en collaboration avec l'équipe JXTA de Sun Microsystems. Notre contribution dans ce domaine est d'avoir caractérisé le comportement d'un protocole de gestion du réseau virtuel JXTA. Un article sur ce sujet est en cours de rédaction.

Langage de description d'applications basées sur la plate-forme JXTA.
L'évaluation et l'utilisation de bibliothèques P2P sur les grilles de calcul pose également le problème de leur déploiement. En effet, ces systèmes sont habituellement déployés par leurs utilisateurs. Cela n'est toutefois pas possible sur les grilles de calcul en raison de leur objectif de transparence d'utilisation. Notre contribution consiste en la proposition d'un langage de description, appelé JDL (JXTA *Description Langage*), pour le déploiement automatique d'applications basées sur la plate-forme JXTA et s'exécutant sur les grilles de calcul. Cette proposition est accompagnée de sa mise en

[2] Maître de conférences à l'ENS Lyon au sein du projet GRAAL.
[3] Chargé de recherche à l'INRIA au sein du projet PARIS.
[4] À l'époque étudiant à l'université du New Hampshire, sous la direction du Professeur Philip Hatcher.

œuvre dans l'outil de déploiement générique ADAGE. Elle a été réalisée en étroite collaboration avec Christian Pérez. Elle a été validée par le déploiement à grande échelle de réseaux virtuels JXTA permettant de caractériser le comportement d'un protocole JXTA. Elle a par ailleurs permis de faciliter le déploiement de JUXMEM en vue de son évaluation.

Les expérimentations impliquées dans l'ensemble de ces contributions ont été réalisées sur la grille expérimentale Grid'5000. Nous avons donc crucialement bénéficié de l'excellent travail de l'ensemble des personnes[5] qui a contribué à la mise en place et au bon fonctionnement de cette infrastructure.

Publications

Les travaux que nous présentons dans ce manuscrit ont été publiés dans diverses conférences et revues. Ces publications concernent :

- le concept de service de partage de données [1, 2, 9, 10] ;
- son utilisation et évaluation dans le modèle de programmation Grid-RPC [1] ;
- son utilisation et évaluation dans les modèles de programmation à base de composants [3, 4] ;
- l'évaluation et l'amélioration des performances des couches de communication des implémentations de référence de JXTA [5, 6, 12]
- le déploiement à grande échelle d'applications basées sur JXTA [7].

Ces différents articles ont en outre fait l'objet de rapports de recherche INRIA et IRISA.

Chapitres de livres

[1] Gabriel ANTONIU, Marin BERTIER, Eddy CARON, Frédéric DESPREZ, Luc BOUGÉ, Mathieu JAN, Sébastien MONNET, and Pierre SENS. « *Future Generation Grids* », Chapter GDS: An Architecture Proposal for a Grid Data-Sharing Service, pages 133–152. CoreGRID series. Springer-Verlag, 2006.

Articles dans des revues internationales

[2] Gabriel ANTONIU, Luc BOUGÉ, and Mathieu JAN. « JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid ». *Scalable Computing: Practice and Experience*, 6(3):45–55, September 2005.

Conférences internationales avec comité de lecture

[3] Gabriel Antoniu, Hinde Lilia Bouziane, Landry Breuil, Mathieu Jan, and Christian Pérez. « Enabling Transparent Data Sharing in Component Models ». In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '06)*, pages 430–433, Singapore, May 2006.

[5] Et plus particulièrement des ingénieurs du site de Rennes pour leur support « local ».

[4] Gabriel Antoniu, Hinde Lilia Bouziane, Mathieu Jan, Christian Pérez, and Thierry Priol. « Combining Data Sharing with the Master-Worker Paradigm in the Common Component Architecture ». In *Proceedings of the Joint Workshop on HPC Grid programming Environments and COmponents and Component and Framework Technology in High-Performance and Scientific Computing (HPC-GECO/CompFrame 2006)*, pages 10–18, Paris, France, June 2006. Held in conjunction with the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC 15).

[5] Gabriel Antoniu, Mathieu Jan, and David A. Noblet. « Enabling the P2P JXTA Platform for High-Performance Networking Grid ». In *Proceedings of the 1st International Conference on High Performance Computing and Communications (HPCC '05)*, number 3726 in Lecture Notes in Computer Science, pages 429–440, Sorrento, Italy, September 2005. Springer.

[6] Gabriel Antoniu, Philip Hatcher, Mathieu Jan, and David A. Noblet. « Performance Evaluation of JXTA Communication Layers ». In *Proceedings of the Workshop on Global and Peer-to-Peer Computing (GP2PC 2005)*, Cardiff, UK, May 2005. Held in conjunction with the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '05). Best presentation award.

[7] Gabriel Antoniu, Luc Bougé, Mathieu Jan, and Sébastien Monnet. « Large-scale Deployment in P2P Experiments Using the JXTA Distributed Framework ». In *Euro-Par 2004: Parallel Processing*, number 3149 in Lecture Notes in Computer Science, pages 1038–1047, Pisa, Italy, August 2004. Springer.

[8] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. « JuxMem: An Adpative Supportive Platform for Data Sharing on the Grid ». In *Proceedings of the ACM Workshop on Adaptive Grid Middleware (AGridM 2003)*, pages 49–59, New Orleans, Louisiana, September 2003. Held in conjunction with PACT 2003.

[9] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. « Peer-to-Peer Distributed Shared Memory? ». In *Proceedings of the IEEE/ACM 12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003), Work in Progress Session*, pages 1–6, New Orleans, Louisiana, September 2003.

Conférences nationales avec comité de lecture

[10] Gabriel Antoniu, Luc Bougé et Mathieu Jan. « La plate-forme JuxMem : support pour un service de partage de données sur la grille ». Dans *Actes des Rencontres francophones du parallélisme (RenPar 15)*, pages 145–152, La Colle-sur-Loup, octobre 2003.

Actes d'écoles sans comité de lecture

[11] Mathieu Jan. « JuxMem : un service de partage de données modifiables pour les grilles ». Dans *DistRibUtIon de Données à grande Echelle (DRUIDE 2004)*, pages 59–70, Le Croisic, France, mai 2004. IRISA.

Rapports de Recherche non publiés

[12] Gabriel ANTONIU, Mathieu JAN, and David A. NOBLET. « A practical example of convergence of P2P and grid computing: an evaluation of JXTA's communication performance on grid networking infrastructures ». Research Report RR-5718, INRIA, IRISA, Rennes, France, November 2005.

[13] Gabriel ANTONIU, Luc BOUGÉ, and Mathieu JAN. « JuxMem: Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service ». Research Report RR-5082, INRIA, IRISA, Rennes, France, January 2004.

Organisation de ce manuscrit

Ce manuscrit s'articule en quatre parties. La première partie introduit notre contexte d'étude et les différents travaux existants. Le chapitre 2 présente le contexte de nos travaux : les grilles de calcul. Le chapitre 3 pose la problématique de la gestion des données sur les grilles de calcul, présente un panorama des solutions dans ce domaine et souligne leurs limites. Le chapitre 4 présente deux autres approches pour la gestion de données dans les systèmes distribués : les systèmes à mémoire virtuellement partagée (MVP) et les systèmes pair-à-pair (P2P).

Dans la deuxième partie de notre manuscrit, nous introduisons notre contribution pour la gestion des données sur les grilles de calcul. Le chapitre 5 définit le concept de service de partage de données, spécifie son interface de programmation et propose une architecture appelée JUXMEM. La conception de cette architecture suit un modèle P2P et sa mise en œuvre utilise la plate-forme P2P JXTA. Cela motive la présentation, dans le chapitre 6, des principaux concepts et protocoles définis par JXTA. Enfin, le chapitre 7 présente notre implémentation de JUXMEM au-dessus de JXTA, ainsi qu'une micro-évaluation de celle-ci.

La troisième partie de ce manuscrit présente l'évaluation de JUXMEM dans différents modèles de programmation utilisés sur les grilles de calcul. Le chapitre 8 décrit l'intégration et l'évaluation du concept de service de partage de données dans le modèle de programmation Grid-RPC. Le chapitre 9 introduit notre modèle de port de données, qui correspond à l'intégration du concept de service de partage de données dans les modèles à base de composants.

La quatrième et dernière partie de ce manuscrit décrit nos contributions à la plate-forme P2P JXTA. Notre objectif premier est d'adapter JXTA aux grilles de calcul pour un meilleur support de JUXMEM. Toutefois, ces contributions ont une portée plus générale et peuvent servir à d'autres projets de recherche visant à utiliser les techniques P2P sur les grilles de calcul. Le chapitre 10 présente donc l'évaluation et les améliorations des performances des couches de communication de JXTA que nous avons obtenues. Le chapitre 11 propose un langage de description d'applications basées sur la plate-forme JXTA en vue de leur déploiement sur une grille.

Enfin, le chapitre 12 rassemble ces contributions, en souligne les limites et présente les perspectives qu'ouvrent nos travaux.

Première partie

Contexte d'étude : la gestion de données à grande échelle

Chapitre 2

Les grilles de calcul

Sommaire

Dans cette première partie du manuscrit, nous introduisons notre contexte d'étude. Nous présentons tout d'abord les infrastructures d'exécution que nous visons : les grilles de calcul. Puis, le chapitre 3 présente un panorama des solutions utilisées pour la gestion des données sur de telles infrastructures. Enfin dans le chapitre 4, nous replaçons ce problème dans le cadre plus large de la gestion de données dans les systèmes distribués, en présentant deux approches différentes.

L'objectif de ce chapitre est de mieux cerner le terme *grille de calcul*, en précisant notamment les propriétés de ce type d'infrastructure. Nous commençons par en mentionner les origines. Nous en présentons ensuite deux exemples : la grille expérimentale Grid'5000 [51] et la grille de calcul américaine TeraGrid [140]. Puis, nous donnons un aperçu des intergiciels typiquement utilisés pour leur gestion ainsi que des applications scientifiques s'exécutant sur ces infrastructures. Enfin, nous analysons les potentialités de l'utilisation de techniques pair-à-pair (P2P) dans le cadre des grilles de calcul.

2.1 Origine et définitions

L'augmentation des débits des réseaux longue distance, en anglais *Wide-Area Networks* (WAN), a permis d'envisager l'exécution d'applications distribuées sur des machines réparties dans différents lieux géographiques. Le précurseur dans ce domaine est le réseau *I-Way* [69] qui en 1995 interconnecte 17 centres de calcul américains et constitue ainsi en quelque sorte l'ancêtre des grilles de calcul[1].

Le terme *grille de calcul*, en anglais *computing grid*, est pour la première fois utilisé en 1998 dans l'ouvrage fondateur de Ian Foster et Karl Kesselman [85]. Ce terme a été retenu par analogie avec les systèmes de distribution d'électricité (en anglais *electric power grid*), tels que ceux d'Électricité de France (EDF [204]) par exemple. En effet, l'objectif à terme des grilles de calcul est de fournir un ensemble de ressources informatiques virtuellement infini pour le calcul et le stockage à travers une banale « prise grille », c'est-à-dire de manière transparente pour les utilisateurs. La localisation de ces ressources informatiques est géographiquement distribuée à travers d'une ou plusieurs nations. La définition des grilles de calcul proposée par Foster et Kesselman est la suivante :

Définition 2.1 : ***grille de calcul*** — *Infrastructure matérielle et logicielle qui fournit un accès fiable, cohérent, accessible de n'importe où, peu cher et avec des capacités de calcul haut de gamme.*

En raison de son caractère relativement vague, d'autres définitions ont été proposées [92, 88, 83]. Toutes s'accordent toutefois pour énoncer que les grilles de calcul permettent de résoudre des problèmes complexes par la mise en commun de ressources informatiques (disques, processeurs, etc.)[2]. Par ailleurs, une classification souvent employée distingue deux catégories de grilles : les *grilles de calcul* (mise en commun de machines dédiées aux calculs) et *grilles de données* (mise en commun de machines dédiées au stockage d'énormes quantités de données, telles que des bases de connaissances dans un domaine). Les *grilles de vol de cycles* (en anglais *desktop grids* ou *cpu scavenging grids*) sont généralement incluses dans les grilles de calcul. Elles consistent en la mise en commun, la nuit, de machines de bureau utilisées dans la journée. La récupération de cycles de calcul alors réalisée permet d'utiliser ces machines pour exécuter des tâches diverses. Toutefois, la distinction entre les grilles de calcul et les grilles de données est floue et tend à s'estomper aujourd'hui [28]. En effet, les grandes bases de connaissances, issues des grilles de données, sont de plus en plus souvent accédées par des applications s'exécutant sur des grilles de calcul. Cependant, nous tenons à préciser que ce sont bien les grilles de calcul, et non les grilles de données, qui constituent le contexte de nos travaux, même si ce manuscrit se concentre sur la gestion des données. En effet, nous traitons de la gestion des données partagées par des applications s'exécutant sur des grilles de calcul.

La multiplicité des définitions d'une grille de calcul nous amène donc à préciser notre propre vision des ces infrastructures. Nous pouvons constater que *l'infrastructure matérielle* et *l'infrastructure logicielle* de ces environnements sont *mélangées* dans la définition 2.1. Notre vision consiste à séparer ces deux points en définissant d'un côté le terme *grilles de calcul* pour l'infrastructure matérielle, et de l'autre côté le terme *intergiciel pour grilles de calcul* pour l'infrastructure logicielle. Dans la suite de ce chapitre et par souci de concision, le terme *calcul*

[1] L'appellation exacte de ce type de déploiement est *metacomputing*.

[2] Dans la suite de ce chapitre les termes *ressources* et *ressources informatiques* sont utilisées de manière indifférenciée.

est omis lorsque nous parlons des intergiciels pour grilles. Dans cette section, nous donnons notre définition des grilles de calcul et nous en présentons des exemples à la section 2.2. Puis, la section 2.3 définit le terme intergiciel pour grilles et en présente des exemples parmi les plus représentatifs.

Une grille de calcul étant un cas particulier de système distribué, nous reproduisons la définition telle qu'elle est énoncée par Andrew Tanenbaum dans [172].

Définition 2.2 : ***système distribué*** — Collection de ressources[3] indépendantes qui apparaissent à l'utilisateur comme un seul et unique système cohérent.

Nous définissons alors les grilles de calcul de la manière suivante.

Définition 2.3 : ***grille de calcul*** — Système distribué constitué de l'agrégation de ressources réparties sur différents sites et mises à disposition par plusieurs organisations différentes. Chaque site dispose d'un grand nombre de ressources.

La dernière phrase de la définition 2.3 impose une contrainte forte sur la localisation des ressources. Elle permet de distinguer si un système distribué est une grille de calcul ou non. Par ailleurs, la définition 2.3 est très large. L'objectif est d'englober les différents types de grilles existants par l'intermédiaire du terme *ressources*. Dans notre contexte d'étude les ressources partagées sont généralement des machines de type standard (PC). Toutefois, le cas général peut par exemple inclure des capteurs, des outils de visualisations, des dispositifs de contrôle et de surveillance, etc.

Notre définition impose une contrainte sur la distribution des ressources partagées : celles-ci doivent être réparties sur différents lieux géographiques, également appelés *sites*.

Définition 2.4 : ***site*** — Lieu géographique regroupant un ensemble de ressources informatiques administrées de manière autonome et uniforme.

Les sites d'une grille de calcul peuvent être organisés de manières différentes, par exemple, sous la forme d'une (ou d'une collection de) grappe(s) de machines ou de supercalculateurs, voire un mélange des deux. Nous nous intéressons plus particulièrement au cas des grilles de calcul constituées de sites disposant de grappes de machines.

Définition 2.5 : ***grappe de machines (en anglais cluster)*** — Regroupement de machines indépendantes interconnectées par un même équipement réseau[4], localisées dans un même site et disposant d'au moins une machine spécifique de gestion appelée *machine frontale*.

Notons que cette machine frontale se doit d'offrir un service de gestion des ressources disponibles dans une grappe de machines. Généralement, les machines qui composent une grappe de machines ont des caractéristiques matérielles homogènes (processeurs, mémoire vive, technologies d'interconnexion réseau, etc.). Les grappes de machines peuvent être classifiées en différentes catégories selon qu'elles visent la haute disponibilité, l'équilibrage de charge ou la haute performance. Dans ce dernier cas, les machines sont alors interconnectées par un réseau spécialisé, appelé *System Area Network* (SAN) en anglais. Ces réseaux offrent des débits allant jusqu'à 20 Gb/s et une latence de l'ordre de quelques microsecondes. Des

[3] La définition originelle utilise le terme *ordinateurs*.

[4] Typiquement un *switch*.

exemples de SAN couramment utilisés sont : Myrinet [230], Infiniband [216], SCI [203] et Quadrics.

Dans notre contexte d'étude, nous considérons le cas particulier où une grille de calcul est une fédération de plusieurs sites, chaque site étant constitué d'une ou plusieurs grappes de machines haute performance.

2.2 Infrastructures matérielles des grilles de calcul

Nous présentons dans cette section les caractéristiques physiques de ces infrastructures, à travers deux exemples représentatifs de grilles de calcul que nous visons. Nous décrivons tout d'abord la grille expérimentale Grid'5000 [51], puis la grille de calcul TeraGrid [140]. Enfin, pour terminer nous généralisons les caractéristiques physiques de ces infrastructures.

Grid'5000. Le projet Grid'5000 consiste en la construction d'une grille expérimentale. En effet, elle diffère des autres grilles de calcul par sa capacité d'être hautement reconfigurable et contrôlable. Par exemple, elle permet à chaque utilisateur de déployer son propre système d'exploitation, via l'outil Kadeploy2 [223]. À terme, Grid'5000 devrait être constituée de 5000 processeurs (actuellement 2600 sur 1244 machines[5]) répartis sur neuf sites français : Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse. L'interconnexion entres les différents sites est assurée par la version 4 du réseau Renater [240] qui fournit un débit[6] de 10 Gb/s. La latence entre les machines de différents sites varie de 4 ms à 29 ms, selon les paires de sites considérées[7]. La figure 2.1 présente l'interconnexion[8] des différents sites de Grid'5000. Les machines des différents sites sont généralement équipées de biprocesseurs Opteron d'AMD. Cette grille est toutefois hétérogène : elle comporte également des machines équipées de processeurs Intel ou PowerPC. Les machines à l'intérieur d'un site sont interconnectées par des réseaux Gigabit Ethernet, parfois Myrinet comme sur les sites de Bordeaux, Grenoble, Orsay, Lyon, Rennes et Sophia-Antipolis, ainsi qu'Infiniband comme sur le site de Rennes.

TeraGrid. Le projet TeraGrid est un projet américain visant à mettre en place la plus grande et la plus puissante grille de calcul scientifique au monde. Environ 22 000 processeurs sont actuellement répartis sur neuf sites : Austin, Bloomington, Boulder, Chicago, Oark Ridge, Pittsburgh, San Diego, Urbana et West Lafayette. Un réseau optique longue distance assure la liaison entre les différents sites et la figure 2.2 en présente l'interconnexion[9]. Ce réseau peut atteindre à certains endroits un débit de 30 Gb/s. La latence entre les machines de différents sites varie de 3 ms à 80 ms, selon les paires de sites considérées. Les machines constituant les différents sites sont très variées : Intel Itanium, Power4 IBM, SGI Altix, etc. Les systèmes d'exploitation sont également variés : Linux, Unix, etc. Enfin, les réseaux locaux utilisés sont Myrinet, Quadrics, Infiniband, Gigabit Ethernet et 10 Gigabits Ethernet.

[5] Nombre relevé en septembre 2006.
[6] Migration de 1 Gb/s à 10 Gb/s réalisée durant l'été 2006.
[7] Cette variation dépend de la distance entre les sites et des équipements traversés.
[8] Chemins physiques réels approximatifs et carte extraite du site Web de Grid'5000 [210].
[9] Informations et figure extraite du site Web du projet TeraGrid [140] en septembre 2006.

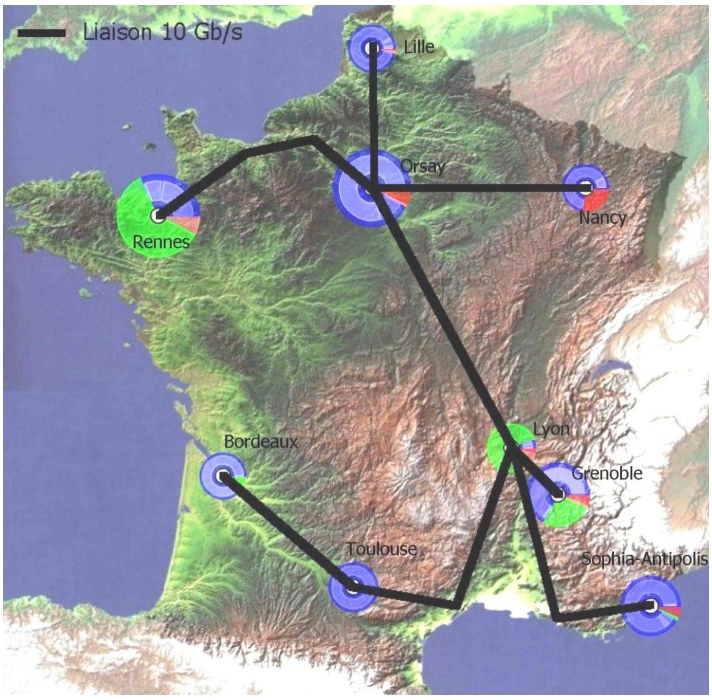

FIG. 2.1 – L'interconnexion des différents sites de la grille expérimentale Grid'5000.

FIG. 2.2 – L'interconnexion des différents sites de la grille de calcul TeraGrid.

D'autres projets de grilles de calcul existent, notamment au niveau international. Nous pouvons par exemple citer : les grilles européennes EGEE [192] (*Enabling Grids for E-science in Europe*) et DEISA [135] (*Distributed European Infrastructure for Supercomputing Applications*) basées sur le réseau d'interconnexion GÉANT2 à 10 Gb/s [209], la grille Asie-Pacifique ApGrid [195], la grille chinoise CNGrid [199], la grille suédoise Swegrid [246] constituée de 600 machines réparties sur 6 sites et basées sur des liaisons à 10 Gb/s entre les sites, la grille norvégienne NorGrid [233], etc. Des contre-exemples de grilles de calcul sont PlanetLab [60] et Emulab [188]. Il s'agit uniquement de systèmes distribués. En effet, chaque site constituant PlanetLab n'héberge généralement qu'une seule machine, alors qu'Emulab n'est localisé que dans un seul site.

En généralisant les caractéristiques de ces infrastructures, nous pouvons considérer qu'une grille de calcul est une *architecture parallèle distribuée*. En effet, l'ensemble des machines d'un site constitue un système parallèle. Dans le cas général il peut s'agir de supercalculateurs ou de grappes de machines haute performance par exemple. Dans notre cas en revanche, nous considérons un site comme un ensemble de n grappes de machines (avec éventuellement $n = 1$). Comme nous l'avons vu à la section précédente, les machines d'une grappe sont reliées par un réseau haute performance (SAN), tel que Myrinet, Infiniband, Quadrics ou SCI. Les sites sont interconnectés entre eux par l'intermédiaire de réseaux longue distance à haut débit, en anglais *Wide Area Network* (WAN). L'ensemble forme donc bien un système distribué. Les réseaux WAN présentent des caractéristiques différentes des SAN. Ils accusent généralement une plus forte latence pouvant atteindre 100 ms, selon le type de médium utilisé utilisé pour interconnecter les machines et/ou si le nombre d'équipements réseaux traversés est important. Nous pouvons donc constater une différence en termes de latence entre deux machines situées à l'intérieur d'un site (noté *intrasite*) et entre deux machines de deux sites différents (noté *intersite*). C'est certainement la caractéristique physique la plus importante des grilles de calcul, puisque le ratio entre les deux valeurs varie entre 2 000 et 50 000. Les réseaux WAN disposent toutefois d'un fort débit pouvant aller jusqu'à plusieurs dizaines de gigabits par secondes, comme c'est le cas dans la grille TeraGrid [140].

Par ailleurs, notons que chaque site d'une grille de calcul est administré de manière autonome vis-à-vis des autres sites. Par exemple, les politiques de sécurité peuvent être différentes d'un site à un autre. Un autre exemple concerne la composition du matériel formant les grappes de machines de chaque site. Chaque site est libre de choisir le type de processeur pour ses machines (Intel, AMD, Sun, IBM, etc.), le réseau d'interconnexion entre les machines (Gigabit Ethernet, Quadrics, Myrinet, Infiniband, etc.), la capacité de stockage des machines, etc. Ainsi, les grilles de calcul sont des infrastructures *hétérogènes*. La gestion décentralisée de ces différents aspects reste cependant coordonnée au niveau de la grille, afin d'assurer la disponibilité des ressources partagées mises à disposition des utilisateurs.

Enfin, une grille de calcul est une infrastructure de grande taille qui peut impliquer jusqu'à plusieurs milliers de machines. En conséquence, ce sont des architectures dynamiques où la perte de machines, voire de sites entiers doivent être considérées comme des événements possibles. Par exemple, sur une période de 6 mois[10], 75 incidents majeurs (perturbations, maintenance et/ou arrêt de sites entraînant une période d'indisponibilité de quelques secondes à plusieurs jours) ont eu lieu sur la grille expérimentale Grid'5000, soit un évé-

[10] De janvier à juin 2006.

nement tous les 2,4 jours en moyenne. Ces informations ont été obtenues par consultation d'une page Web de cette grille [210]. Elles sont certainement en dessous de la réalité car elles ne prennent pas en compte, par exemple, les arrêts momentanés de machines lorsque celles-ci sont en nombre limité. Ce type d'événements, mais également l'ajout de machines voire de sites entiers[11], doivent donc être pris en compte dans la conception de programmes s'exécutant sur les grilles de calcul. Nous signifions cette propriété des grilles de calcul par le terme *volatilité*.

Définition 2.6 : ***volatilité*** — Un élément est dit volatile s'il se comporte de manière conforme à sa spécification jusqu'à subir une défaillance. Au bout d'un certain temps après cette défaillance, l'élément suit à nouveau sa spécification.

Nous ne considérons donc pas qu'un élément qui subit une défaillance définitive est volatile.

Définition 2.7 : ***défaillance (en anglais failure)*** — Événement conduisant à ne plus se comporter conformément à sa spécification.

Dans le contexte des grilles de calcul, une machine est volatile si elle subit une coupure de courant, en raison de la défaillance de la climatisation par exemple et donc nécessitant un arrêt d'urgence des machines pour éviter tout risque de surchauffe.

De par leur hétérogénéité, leur gestion décentralisée et leur échelle, les grilles de calcul sont des infrastructures complexes. En outre, les différentes grilles de calcul mentionnées dans cette section pourraient être connectées entre elles, afin de former des « *grilles de grilles de calcul* ». Ceci décuple encore une fois la puissance offerte par ces infrastructures, mais également leur complexité !

2.3 Intergiciels pour les grilles de calcul

Nous avons montré dans la section précédente que les grilles de calcul sont hétérogènes et complexes. Or, l'objectif des grilles de calcul est de fournir leurs ressources de manière transparente et la plus simple possible. Atteindre un tel objectif nécessite l'utilisation de logiciels spécifiques pour leur gestion, appelés *intergiciels pour grilles*. Définissons tout d'abord le terme intergiciel.

Définition 2.8 : ***intergiciel (en anglais middleware)*** — Logiciel qui se trouve entre l'application de l'utilisateur et les ressources informatiques matérielles et logicielles sous-jacentes, afin d'aider l'application à utiliser ces ressources.

Les objectifs des intergiciels sont variés, par exemple d'assurer la portabilité du code ou cacher l'aspect distribué de l'environnement d'exécution. Dans le cadre des grilles de calcul, les ressources matérielles sont les grilles de calcul, telles que nous les avons décrites à la section 2.2. Les ressources logicielles sont, par exemple, le système d'exploitation qui s'exécute sur chacune des machines d'une grille de calcul. La distinction entre un intergiciel et une bibliothèque utilisée pour la programmation d'applications réside dans la présence ou l'absence d'un *exécutif*. Un exécutif est défini de la manière suivante :

[11] Nous incluons dans ces cas les redémarrages de machines suite à des maintenances.

Définition 2.9 : ***exécutif (en anglais runtime)*** — Parties d'un intergiciel utilisées par les applications à l'exécution.

Par exemple, le JDK de Sun Microsystems est un intergiciel, alors que la machine virtuelle Java (JVM) est un exécutif. Par ailleurs, nous distinguons principalement deux types d'intergiciels pour grilles : les *intergiciels de gestion et d'accès aux ressources* et les *intergiciels pour la programmation d'applications*. Nous détaillons leurs rôles et en donnons des exemples représentatifs dans les deux paragraphes suivants.

Intergiciels de gestion et d'accès aux ressources. Le rôle de ces intergiciels est de donner une vision globale et uniforme des ressources partagées au sein d'une grille de calcul. Ainsi, ils s'attachent à la gestion de plusieurs aspects différents qui sont décrits ci-dessous.

1. L'accès aux différentes ressources informatiques, c'est-à-dire l'allocation des ressources nécessaires pour les applications ainsi que le lancement des applications sur ces ressources.
2. Le placement des données nécessaires en entrée aux applications, ainsi que la récupération des données produites par celles-ci à la fin de leur exécution.
3. La mise en place de services d'information permettant de connaître l'état des ressources mises à disposition par une grille.
4. La mise en place des politiques de sécurité définies par les administrateurs, par exemple la méthode choisie pour l'authentification des utilisateurs.

La réalisation du premier point nécessite des interactions avec les *gestionnaires de ressources* des différentes grappes de machines des sites d'une grille (en anglais *job schedulers*). Le rôle d'un gestionnaire de ressources pour grappe de machines est d'ordonnancer au mieux les différentes tâches soumises par les utilisateurs. Une tâche regroupe un ensemble de calculs, potentiellement lancés sur plusieurs machines et constituant tout ou partie d'une application. Des exemples de gestionnaires de ressources sont PBS [237], LSF [227], SGE [243], OAR [50], Condor [177], etc. Un problème encore non résolu est la gestion *coordonnée* des différents gestionnaires de ressources d'une grille (en anglais *co-scheduling*). L'objectif est d'exécuter de manière simultanée plusieurs tâches, dépendantes entre elles, sur différentes grappes de machines. Des techniques de co-allocation doivent être développées, comme par exemple dans le gestionnaire de ressources de grilles KOALA [126]. Une autre possibilité est d'autoriser les utilisateurs à réserver un ensemble de machines sur différents sites pendant une certaine période de temps, comme cela est possible en utilisant la version grille d'OAR, OARGRID [234]. Des recherches sont en cours à ce sujet.

Au-dessus de ces différents gestionnaires de ressources pour grappes de machines, l'intergiciel grille le plus utilisé pour la gestion et l'accès aux ressources d'une grille de calcul est Globus [86]. Nous le décrivons brièvement ci-dessous.

Globus est une collection d'outils génériques (bibliothèques et outils en ligne de commande) pour la gestion et l'accès aux ressources d'une grille. Il est principalement développé à *Argonne National Laboratory*, sous la direction de Ian Foster. Le module de gestion des ressources de Globus s'appelle GRAM (pour *Globus Resource Allocation Manager*). Il permet de soumettre des tâches sur des ressources distantes, ainsi que de contrôler ces tâches. Pour ce faire, un script RSL (*Resources Specification Language* [241]) doit être écrit par les utilisateurs afin de spécifier les fichiers exécutables à lancer, les

données à utiliser, etc. À partir de ce fichier, Globus génère l'ensemble des requêtes nécessaires aux multiples gestionnaires de ressources (potentiellement de types différents) des différentes grappes de machines de la grille utilisée. Une politique d'allocation simultanée d'ensembles de ressources est disponible dans Globus, sous le nom de DUROC (*Dynamically Updated Request Online Co-allocator*). Le module de gestion des données de Globus est présenté plus en détail à la section 3.2.1. Le service d'information de Globus est appelé MDS (*Monitoring & Discovery System*). Il permet par exemple d'obtenir des informations sur l'état des ressources d'une grille. Enfin, le module GSI (*Grid Security Infrastructure*) permet l'authentification des utilisateurs grâce à des certificats.

Globus sert de brique de base à un ensemble d'intergiciels commerciaux pour les grilles, tels que *Univa Globus Enterprise* [247], *Platform Globus Toolkit* [238] et *Moab Grid Suite* [229]. Il sert également de fondation à différents intergiciels d'origine académique, tels que ceux déployés sur la grille DataGrid [77], le projet de grille et d'intergiciel GridLab [211] et l'intergiciel Gridbus [48]. Dans les deux cas, les intergiciels développés intègrent tout ou partie des services de Globus, et parfois offrent de nouveaux services. Par ailleurs, d'autres intergiciels pour grilles existent. Nous en listons brièvement quelques-uns ci-dessous.

UNICORE (*UNiforme Interface to COmputing REsources* [16]) est un projet européen, d'origine allemande, visant à permettre aux scientifiques d'exploiter les différents centres de calcul répartis en Europe. Contrairement à Globus, UNICORE est un logiciel intégré avec des interfaces graphiques. Toutefois, il fournit les mêmes fonctionnalités que Globus. Cet intergiciel a servi de base pour le développement de l'intergiciel japonais pour grille NAREGI [125].

Legion [92] a pour objectif de donner la vision d'un unique supercalculateur virtuel par agrégation des ressources de la grille. Pour atteindre ce but, Legion utilise une approche originale où toutes les ressources (matérielles, logicielles, etc.) sont des objets. Ils sont décrits par un langage de description d'interface, et les communications entre les différentes instances des objets utilisent le modèle de programmation par appel de procédures (*Remote Procedure Call*, RPC). Ce projet vise la gestion de centaines de machines et de millions d'objets. Les différentes interfaces des objets définis par Legion sont spécialisables, héritables, etc. afin de rendre le système extensible.

Condor-G [90] offre une interface similaire au gestionnaire de ressources Condor [177] au-dessus de Globus. Il vise à mettre en œuvre la récupération de cycles des machines non utilisées, et permet le déploiement d'une application simple (un seul programme) sur une seule grappe de machines en utilisant un mécanisme de correspondance des caractéristiques des ressources avec les contraintes des applications (en anglais *Match-Making*). Il utilise Globus pour le lancement des tâches (d'où le « G »).

Ces différents intergiciels ne sont malheureusement pas interopérables entre eux. Ainsi, une grille de calcul est souvent étiquetée comme une grille Globus, UNICORE, Legion, etc. Toutefois, le *Global Grid Forum* (GGF) a établi une norme appelée *Open Grid Services Architecture* (OGSA [87]) afin de promouvoir l'interopérabilité de ces différents systèmes et donc des grilles de calcul. L'objectif du GGF, à travers ses nombreux groupes de travail et de recherche[12], est de promouvoir l'utilisation des grilles par la création de cette spécification OGSA. Cette norme a été implémentée dans les versions 3 et 4 de Globus, mais également

[12]Plus précisément 29 groupes de travail et 22 groupes de recherche.

par NAREGI [125]. La version 3 de Globus est une implémentation de OGSA basée sur *Open Grid Services Infrastructure* (OGSI [182]), alors que la version 4 repose sur *Web Service Resource Framework* (WSRF [62]). La motivation principale de ce dernier changement de choix d'implémentation est la convergence des *Web services* avec les *Grid services*, ne justifiant plus l'utilisation d'un canevas (en anglais *framework*) de programmation spécifique pour le développement des intergiciels pour grilles. Pour plus de détails sur les *Web services*, les *Grid services* et les raisons de ce choix, le lecteur intéressé peut se reporter à [63].

La principale limite des intergiciels de gestion et d'accès aux ressources des grilles est généralement d'offrir des services de relativement bas niveau. Ainsi, des outils offrant un niveau d'abstraction supérieur sont actuellement développés afin de faciliter le déploiement d'applications sur les grilles de calcul. Pour ce faire, ils offrent un (ou des) modèle(s) de description des applications, afin de décharger l'utilisateur de la nécessité de maîtriser les différents intergiciels pour grilles. Le modèle de description d'une application est généralement guidé, voire imposé, par le (ou les) intergiciel(s) de programmation utilisé(s) pour la conception de cette application. Nous pouvons par exemple citer les outils de déploiement spécifiques à un intergiciel de programmation tels que :

- GoDIET [53] pour l'intergiciel Grid-RPC DIET ;
- JDF [217] pour l'intergiciel P2P JXTA-J2SE ;
- ProActive [31] pour le modèle de composant du même nom ;
- Pegasus (*Planning for Execution in Grids [68]*) pour les applications à base de flux de données (en anglais *workflow*) ;

À l'opposé, l'outil ADAGE [114] est un outil générique pour la description de différents types d'applications, s'appuyant sur un ou plusieurs intergiciels de programmation (CCM [136], MPICH1-P4 [93], MPICH-G2 [104], GRIDCCM [145] et JXTA [181]). Il est décrit plus en détail à la section 11.2.3.

Intergiciels pour la programmation d'applications. L'objectif des intergiciels pour la programmation d'applications est d'offrir des paradigmes de programmation, parfois spécifiques, mais de haut niveau, aux développeurs d'applications. Ainsi, ceux-ci peuvent se concentrer sur la logique de leur application (appelé *code métier*), le cycle de vie de l'application étant entièrement géré par l'intergiciel utilisé. Contrairement aux intergiciels précédents, ceux-ci n'obligent pas le développeur à connaître parfaitement les différents éléments qui composent une grille de calcul et à interagir avec eux. De nombreux intergiciels pour la programmation d'applications existent, parmi lesquels nous pouvons citer les suivants.

- CORBA [137] et ses utilisations pour bâtir des intergiciels de plus haut niveau pour la programmation d'applications. L'intergiciel Grid-RPC DIET en est un exemple typique ainsi que les implémentations du modèle de composant CCM [136], telles que OpenCCM [236] et MicoCCM [228] par exemple.
- Les intergiciels implémentant des modèles composant tels que Fractal [45, 44], Iceni [91], Grid.it [14] et Darwin [121] par exemple ;
- Les intergiciels implémentant le modèle Grid-RPC tels que DIET [54], Ninf [130] et NetSolve [26] par exemple ;
- Les intergiciels de récupération ou vol de cycles de calcul sur les machines non-utilisées, tels que *Berkeley Open Infrastructure for Networking Computing* (BOINC [21]) et XtremWeb [52].

– Les intergiciels offrant le paradigme de programmation par passage de messages. Le modèle le plus connu est la norme MPI (*Message-Passing Inteface*) en version 1.0 [81] et 2.0 [82]. Des implémentations adaptées aux grilles de calcul ont été réalisées, telles que MPICH-G2 [104] et MPICH-V [43].

Notons, toutefois que les intergiciels de gestion et d'accès aux ressources d'une grille ne supportent généralement qu'un unique modèle de programmation. Par exemple, Condor-G s'intéresse principalement au lancement d'applications parallèles écrites en passage de message, que ce soit avec PVM [165] ou MPI [81]. De manière générale, le modèle de programmation le plus souvent supporté par les intergiciels de gestion et d'accès aux ressources des grilles de calcul est MPI.

Systèmes d'exploitation pour grilles de calcul. L'objectif des intergiciels de gestion et d'accès aux ressources étant de fournir l'illusion d'un unique supercalculateur, plusieurs projets visent à modifier directement les systèmes d'exploitation des machines. L'objectif est alors de construire ce qui peut s'appeler un *système d'exploitation pour grilles de calcul*, en y incluant des fonctionnalités jusque-là incluses dans les intergiciels pour grilles. Par exemple, les mécanismes de découverte et d'allocation de ressources ou de tolérance à la volatilité, tels que les points de reprise, peuvent être intégrés à un système d'exploitation pour grilles. L'avantage de ce type de système est alors de fournir ces fonctionnalités de manière transparente pour les applications. Les applications ne doivent pas être portées d'un intergiciel à un autre, les fonctionnalités étant directement incluses dans le système d'exploitation et donc fournies par celui-ci. En outre, les systèmes d'exploitation pour grilles permettent de factoriser des mécanismes en partie similaires, redéveloppés à chaque fois par les différents intergiciels pour grilles qui existent. Les projets WebOS [183][13], 9grid [124] (fondé sur Plan 9 [142]), GridOS [110, 139][14] et Vigne [149] suivent cette approche. Toutefois, la notion de système d'exploitation pour grilles n'est pour l'instant pas encore bien définie, comme le prouve le démarrage en juin 2006 du projet européen XtreemOS [248]. Ce projet de quatre ans vise à définir et concevoir un tel système. Par exemple, l'ensemble des services nécessaires au fonctionnement d'un système d'exploitation pour grilles n'est pas encore défini à ce jour.

Notons que la réalisation de systèmes d'exploitation pour voir un ensemble de machines comme un unique supercalculateur a déjà été effectuée, mais à l'échelle de grappes de machines uniquement. De tels systèmes sont appelés des *systèmes à image unique* (en anglais *Single System Image*). Des exemples de projets actuels qui visent cet objectif sont par exemple Kerrighed [184], OpenMosix [47] et OpenSSI [186]. Ainsi, la mise en place de systèmes d'exploitation pour grilles de calcul peut s'appuyer sur ces systèmes, en supposant leur présence sur potentiellement une partie des grappes de machines des sites d'une grille. C'est notamment une des hypothèses faites par le projet XtreemOS.

2.4 Utilisation des grilles de calcul pour les applications scientifiques

Dans cette section, nous présentons les applications scientifiques typiques qui s'exécutent sur les grilles de calcul. Nous définissons au préalable différents termes que nous

[13] Ce projet ne vise pas spécifiquement les grilles, mais les systèmes distribués.

[14] Deux projets différents sous ce même nom.

utilisons dans la suite de ce manuscrit.

Définition 2.10 : ***application*** — Ensemble de processus qui ont comme objectif la résolution d'un même problème.

Définition 2.11 : ***processus (ou processus applicatif)*** — Ensemble de flots d'instructions (appelés processus légers, en anglais *threads*) exécutés par un processeur. Plusieurs processus peuvent être en cours d'exécution sur une même machine.

Un flot d'instructions est généré par un *compilateur* à partir d'un *code source*, également appelé programme. Ce flot d'instructions est adapté par le compilateur à l'architecture de la machine d'exécution.

L'ensemble des processus qui forment une application peuvent être répartis sur une ou plusieurs machines. Dans notre contexte, ils peuvent être répartis sur plusieurs machines de différents sites d'une grille de calcul. On parle *d'application concurrente* lorsque plusieurs processus, ou un processus composé d'au moins deux processus légers, s'exécutent de manière simultanée et coopèrent pour atteindre leur objectif commun. Une application concurrente est dite *parallèle* lorsque ses tâches sont découpées en sous-tâches pour être exécutées de manière simultanée. Généralement, l'objectif est alors de minimiser le temps global d'exécution de l'application. Un exemple typique est donné par une application composée de plusieurs processus légers (en anglais *multithreaded application*) qui s'exécutent sur une machine physique. Une application concurrente est dite *distribuée*, lorsque la répartition géographique de ses tâches est prépondérante. Un exemple typique est offert par une application client-serveur où un *client* demande à un *serveur* d'exécuter pour lui un *service*. On peut également citer les applications pair-à-pair (P2P). Enfin, une application concurrente peut être à la fois parallèle et distribuée. On parle alors d'application *mixte*.

Les applications scientifiques sont la principale motivation de développement des grilles de calcul. En effet, ces applications sont toujours plus exigeantes en termes de ressources informatiques à utiliser (puissance de calcul, capacité de stockage, etc.). Les principaux avantages pour les applications scientifiques à s'exécuter sur les grilles de calcul sont alors, par exemple, de :

- réaliser des simulations numériques sur un nombre toujours grandissant de paramètres afin d'être de plus en plus précises ;
- réaliser un plus grand nombre d'exécutions indépendantes en parallèle.

Dans notre contexte, les applications dites de *couplage de code* correspondent à la première catégorie d'applications. Dans de telles applications, différents programmes éventuellement parallèles sont exécutés sur potentiellement plusieurs sites d'une grille de calcul. Un exemple d'un tel type d'application est HydroGrid [215]. Cette application modélise et simule les transferts de fluides et de transport de solutés dans des milieux géologiques souterrains. Pour ce faire, elle fait appel à plusieurs phénomènes physico-chimiques, simulés chacun par un code spécifique et parallèle. Les différents codes correspondants doivent alors être exécutés sur différents sites d'une grille, compte tenu de leurs besoins important en termes de ressources de calcul. Afin d'avancer dans leurs simulations respectives et résoudre le problème commun de modélisation, ces codes doivent échanger des données de temps à autre. Un autre exemple d'application de couplage de code est issu du projet EPSN [79] qui offre un Environnement pour le Pilotage de Simulations Numériques distribuées. Cet environnement permet de visualiser les résultats intermédiaires d'une simulation afin de la

piloter. De manière générale, du fait de l'utilisation de *plusieurs* et non d'un code de simulation, ces applications ont besoin d'une infrastructure matérielle d'exécution telle que les grilles de calcul.

La deuxième catégorie d'applications correspond à des applications dites *multiparamétriques*. Une telle application consiste en un ensemble de tâches indépendantes relativement simples, mais qui peuvent être parallèles et nécessiter plusieurs machines. L'avantage de l'utilisation des grilles de calcul est alors de distribuer ces tâches sur un nombre toujours plus grand de machines afin d'augmenter le degré de parallélisme de l'application. Elles ne diffèrent entre elles que par de légers changements, par exemple une valeur différente pour un paramètre d'entrée de l'algorithme utilisé pour effectuer le calcul. L'application Grid-TLSE [67] est un exemple d'application multiparamétrique. Elle permet de tester de nombreux algorithmes de résolution de systèmes linéaires, en faisant varier leurs multiples paramètres. Les données qui entrent en jeu sont des matrices pouvant aller jusqu'à plusieurs centaines de mégaoctets, qu'il faut déplacer et partager entre les différents serveurs de calcul. L'application Grid-TLSE est décrite plus en détail à la section 8.2.2.

2.5 Convergence entre les intergiciels pour grilles et les environnements pair-à-pair

Dans cette section, nous discutons de la convergence des intergiciels pour grilles avec les systèmes P2P [84, 170]. Plus précisément, nous présentons la tendance actuelle à utiliser des techniques P2P pour la conception d'intergiciels pour grilles.

En effet, ces deux types de systèmes visent un même objectif : la mise en commun de ressources informatiques. Leur principale différence réside dans les caractéristiques des infrastructures qu'ils visent. Alors que les systèmes P2P visent l'échelle d'Internet, c'est-à-dire des millions de machines connectées avec intermittence et appartenant généralement à des particuliers, les intergiciels pour grilles visent eux actuellement une échelle entre 1 000 et 10 000 machines disponibles par l'interconnexion de plusieurs grappes de calcul, appartenant à différentes institutions. La conséquence est une plus grande volatilité (voir définition 2.6) des entités constituant un système P2P, par rapport aux entités constituant un intergiciel pour grille. Par ailleurs, les caractéristiques réseaux des grilles de calcul sont très différentes d'Internet. Un système P2P s'exécute généralement sur une topologie réseau plate en latence (de l'ordre de plusieurs dizaines de millisecondes), alors que les grilles de calcul que nous considérons[15] ont une topologie réseau hiérarchique en termes de latence (voir section 2.2). Enfin, l'hétérogénéité déjà présente dans les grilles de calcul est largement accentuée dans un système P2P. La diversité des machines (en termes de matériel : processeur, mémoire, etc.) constituant Internet est plus grande par rapport à des machines constituant une grille. En effet, les machines formant une grille de calcul sont généralement achetées dans une période de temps réduite, typiquement de 1 à 3 ans, alors que les machines formant Internet peuvent avoir des différences d'âge plus importantes.

Malgré la contrainte de volatilité, les systèmes P2P ont démontré leur capacité de passage à l'échelle. Par exemple, les réseaux P2P de partage de fichiers tels que KaZaA [224] ou EDonkey/Overnet [205] supportent plusieurs millions d'utilisateurs. Cette propriété de

[15] C'est à dire des fédérations de grappes de machines haute performance distribuées sur plusieurs sites.

passage à l'échelle manque aux intergiciels pour grilles. Ainsi, à mesure que l'échelle des grilles de calcul augmente, l'utilisation de techniques P2P pour la conception d'intergiciels pour grilles est naturelle. L'objectif est de tirer parti des capacités des systèmes P2P en termes de support de la volatilité et de passage à l'échelle dans les intergiciels de gestion et d'accès aux ressources des grilles de calcul. Zorilla [75] est le parfait exemple d'utilisation de techniques P2P dans la mise en œuvre d'un intergiciel pour les grilles. Il s'agit d'un gestionnaire de ressources qui utilise pour sa mise en œuvre la table de hachage distribuée (DHT) Bamboo [148], issue des recherches du domaine des systèmes P2P. Un autre exemple est le projet SP2A [19] qui utilise JXTA [181] pour mettre en œuvre des algorithmes de recherche de services P2P sur les grilles de calcul. À terme, ce projet offrira un intergiciel de gestion et d'accès aux ressources pour grilles de calcul, bâti sur JXTA.

2.6 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté les origines des grilles de calcul et donné notre définition de ce terme. Puis à titre d'exemple, nous avons décrit deux grilles de calcul : la grille expérimentale Grid'5000 [210, 51] et TeraGrid [140], avant de généraliser les caractéristiques des ces infrastructures. Ensuite, nous avons défini et présenté les différents intergiciels utilisés sur les grilles de calcul. Nous avons également décrit les applications scientifiques qui motivent les recherches dans le domaine des grilles de calcul. Enfin, nous avons justifié l'utilisation de techniques P2P pour la conception d'intergiciels pour grilles.

L'analogie avec les réseaux de distribution d'énergie est cependant loin d'être complète pour les grilles de calcul. Actuellement un utilisateur ne peut pas simplement « brancher » son application sur une grille de calcul. Il doit se soucier de détails techniques et donc connaître les intergiciels pour grilles utilisés. En particulier, la gestion des données au sein de ces environnements est de trop bas niveau. Le programmeur doit explicitement localiser les données, les transférer d'un site à un autre, gérer la cohérence des différentes copies d'une donnée, etc. Le chapitre suivant est précisément consacré à la présentation des différentes solutions actuelles pour la gestion de données sur les grilles de calcul.

Chapitre 3

Gestion des données dans les grilles de calcul

Sommaire

Dans ce chapitre, nous présentons un panorama des solutions utilisées pour la gestion des données sur les grilles de calcul, infrastructures qui ont été présentées au chapitre précédent. Nous établissons tout d'abord la problématique de la gestion de données dans les grilles de calcul. Nous présentons ensuite quatre projets typiques : *Globus* [86], *Internet Backplane Protocol* (IBP [33, 143]), *Grid DataFarm* (GFarm [173]) et *Storage Resource Broker* (SRB) [146, 32], ayant chacun une approche différente. À partir de nos critères d'évaluation, nous déterminons leur adéquation avec les besoins des applications scientifiques et leurs limites.

3.1 La problématique

Le chapitre précédent a montré que les grilles de calcul sont des infrastructures qui peuvent offrir une importante puissance de calcul. Toutefois, elles sont également complexes

car constituées de machines diverses et variées. Cependant, les grilles de calcul ont un objectif en termes de *transparence d'utilisation* pour l'utilisation de leurs ressources. La « prise » d'accès à une grille doit être en mesure de minimiser l'impact sur les applications de l'ensemble complexe des mécanismes de gestion interne.

Cette propriété de transparence doit également s'appliquer sur l'accès et la gestion des données nécessaires pour l'exécution des applications sur les grilles de calcul. Il est souhaitable de décharger les développeurs d'applications de ces aspects afin de leur permettre de se concentrer sur le code métier de leurs applications, mais également de limiter les changements d'interface pour les applications existantes. Le développement d'intergiciels pour la gestion de données sur les grilles est donc nécessaire.

Ces intergiciels sont soumis à diverses contraintes. La première est la quantité de données produites. En effet, à mesure que les applications scientifiques qui s'exécutent au-dessus des grilles de calcul effectuent des simulations de plus en plus précises, le volume de données engendrées augmente vertigineusement. Par exemple, l'expérience CMS (*Compact Muon Solenoid* [98]) réalisée grâce au détecteur de collisions (en anglais *Large Hadron Collider* [116]) du CERN (Centre Européen de Recherche Nucléaire) va générer, à partir de 2007 et quotidiennement, des pétaoctects de données à analyser. Dans ce contexte, comment stocker de manière pérenne ces données ?

Une autre contrainte est la nécessité de garder ces données constamment accessibles depuis n'importe quel site d'une grille. Dans ce cas, quand et comment les stocker de manière efficace ? En effet, ces données peuvent être directement accédées par de nombreux physiciens à travers le monde. Mais elles peuvent également servir d'entrée à des applications de couplage de code ou des applications multiparamétriques, telles que nous les avons décrites à la section 2.4.

Dans ces deux types d'applications des données intermédiaires peuvent être produites. Celles-ci sont susceptibles d'être partagées par les différents processus des multiples codes de simulation répartis sur plusieurs sites d'une grille. Par ailleurs et d'après [100], seulement 10 % de données produites par les expériences du CERN sont modifiables. Toutefois, l'ensemble des métadonnées doit être modifiable par différents processus applicatifs. Dans ce cas, comment *partager de manière cohérente* ces données entre les différents processus applicatifs ? En effet, chaque processus a besoin de mettre à jour tout ou partie de la donnée partagée. Supposons qu'un processus A modifie une donnée partagée D. Un processus B doit alors être en mesure de s'assurer de la lecture de la dernière version de la donnée D. Dans notre cas, cela correspond à la valeur écrite par le processus A.

Afin de mettre en œuvre ce partage cohérent, la donnée doit-elle être migrée ou bien répliquée ? La solution choisie est généralement de répliquer la donnée, à la fois pour des raisons de performances mais également de disponibilité. Sur ce dernier point et de manière plus générale, des mécanismes de tolérance aux fautes doivent être mis en place afin de supporter la perte d'une copie de la donnée. En effet, les grilles de calcul sont des infrastructures sujettes à des contraintes de volatilité de par leur échelle (voir section 2.2 pour une estimation sur la grille expérimentale Grid'5000).

Pour terminer, d'autres contraintes viennent encore complexifier le problème. Les grilles de calcul sont des infrastructures hétérogènes. Il faut donc pouvoir être indépendant du format de stockage local d'une machine : 32 ou 64 bits, petit-boutiste (en anglais *little endian*) ou gros-boutiste (en anglais *big endian*). Par ailleurs, la solution mise en œuvre doit passer à

l'échelle en termes de nombre de machines gérées.

3.2 Panorama des solutions proposées

Dans cette section, à partir de quatre projets ayant chacun une approche différente, nous dressons un panorama des solutions proposées pour la gestion des données sur les grilles de calcul. Ce qui caractérise toutes les approches est le fait que l'ensemble des systèmes concernés offre un espace de nommage global. Dans la suite de cette section, nous présentons tout d'abord l'approche basée sur des catalogues de données. Cette approche est suivie par l'intergiciel Globus, qui a été présenté dans son ensemble à la section 2.3. Puis, nous présentons un projet qui développe une approche originale de *réseau logistique* : le projet IBP [143, 33]. Ensuite, nous présentons le système GFarm [173] qui vise à construire un système de fichiers pour la grille. Enfin, nous présentons le système SRB [146, 32] qui vise à unifier l'interface de programmation d'accès à différents média de stockage. Dans les quatre cas, nous terminons notre description par une brève présentation des projets suivant une approche similaire.

Enfin, nous tenons à citer l'existence de solutions commerciales telles que Avaki IEE [197] ou DataSynapse [201]. Elles définissent souvent leur propre pile logicielle et sont en général peu documentées, ce qui les rend difficiles à évaluer. Dans la suite de ce chapitre, nous faisons donc le choix de ne pas présenter de telles solutions.

3.2.1 Gestion à base de catalogues : l'approche Globus

Globus est un ensemble de briques de base génériques pour la construction d'intergiciels pour grilles. Celles-ci doivent donc être spécialisées afin d'être adaptés à une grille de calcul particulière. Dans cette section, nous nous focalisons sur un aspect de l'intergiciel Globus : la gestion des données. À son plus bas niveau, Globus propose un ensemble d'outils pour effectuer les transferts de données entre plusieurs systèmes de stockage, basés sur l'utilisation de *catalogues*. Nous détaillons ces deux points dans la suite de cette section.

Outils pour les transferts de données. Les protocoles classiques FTP et HTTP peuvent être utilisés pour transférer des données entre plusieurs machines. Toutefois, ils ne sont pas adaptés aux grilles de calcul. GridFTP [15] est un protocole pour le transfert sécurisé et fiable de données sur les grilles de calcul. Il est optimisé pour les liaisons WAN à haut débit et s'appuie sur le protocole FTP (*File Transfer Protocol*). Il enrichit ce dernier de flux de données multiples pour des transferts parallèles, de gestion des reprises en cas d'échec, de transferts entre serveurs contrôlés par une tierce entité, etc. Par ailleurs, il s'appuie sur l'infrastructure de sécurité de Globus (GSI) pour l'authentification des utilisateurs au sein d'une grille. UberFTP est un client interactif en mode console de GridFTP. La version modifiée d'OpenSSH pour l'authentification via GSI, appelée GSI-OpenSSH, offre les services GSI-SCP et GSI-SFTP pour les transferts de données. Enfin, l'outil de transfert fiable de fichiers (en anglais *Reliable File Transfer Service*, RFT) est un service OGSA qui permet d'effectuer des transferts en mode asynchrone. RFT supporte les transferts entre des serveurs GridFTP et FTP.

Outre FTP, HTTP, et GridFTP, il existe de nombreux protocoles de transferts de données, tels que Chirp [198], Remote File I/O (RFIO)[1], Data Link Client Access Protocol (DCAP [59]), DiskRouter [108], etc. Ce dernier protocole suit un modèle différent par rapport à GridFTP et ses communications point-à-point. En effet, il permet de cacher les données sur des machines intermédiaires. Ce type de systèmes est présenté à la section 3.2.2. Face à cette multitude de protocoles de transport de données est né le projet Parrot [176]. Il vise à abstraire les différents protocoles de transport et à fournir une interface POSIX pour l'accès et l'utilisation de fichiers au sein des applications. Dernièrement, notons l'utilisation dans des intergiciels pour grilles de BitTorrent [61] comme protocole de transport de données, à la place de FTP [187].

Catalogues pour les données. Globus fournit trois services sous la forme de catalogues, ayant chacun des objectifs différents. Généralement, ceux-ci sont implémentés en utilisant Lightweight Directory Access Protocol (LDAP [97]) ou une base de données, telle que MySQL [231].

Le catalogue de données (en anglais *Metadata Catalog Service*, MCS) est un service OGSA qui permet d'enregistrer des métadonnées de fichiers et d'objets. Ces métadonnées décrivent par exemple la structure d'un fichier. L'outil de localisation des réplicas (en anglais *Replica Location Service*, RLS[2]) est un catalogue qui permet de déterminer la localisation physique des copies d'une donnée à partir de son nom logique. L'objectif de ce service est de favoriser les accès à des copies proches, et ainsi de réduire le temps d'accès aux données. Par ailleurs, il vise également à améliorer la disponibilité des données gérées par Globus, en facilitant leur réplication. Enfin, Globus définit un service, appelé Chimera, qui permet d'enregistrer les démarches nécessaires pour le traitement d'un ensemble de données. Ces informations sont stockées dans ce qui est appelé un *catalogue de données virtuelles*.

Ces différents catalogues ont servi à la construction d'autres outils spécialisés selon les besoins. Par exemple, la grille de données DataGrid [77] s'est appuyée sur Globus pour implémenter ses propres services de gestion de données à base de catalogues [100]. Elle utilise les outils fournis par Globus pour les transferts de données. Elle a ainsi défini dans sa version 1.0 le service Grid Data Management Pilot [152]. GDMP est un catalogue de gestion de copies pour répliquer manuellement des données mais de manière fiable et sécurisée, grâce à GridFTP. Dans sa version 2.0, ce projet a défini[3] le service de localisation des réplicas RLS. Ce catalogue a été bâti à partir de l'environnement permettant la construction de service de localisation de copies appelé Giggle [58]. Il a également défini d'autres services pour la gestion des copies d'une donnée (notamment Reptor [113], qui s'appuie sur RLS) et l'optimisation des accès aux données (appelé Optor [113]) en utilisant des catalogues de données. Lightweight Data Replicator (LDR [226]) est un autre système de gestion de copies de données à base de catalogues qui s'appuie sur GridFTP, RLS et MCS.

Par ailleurs, nous pouvons citer le projet GridLab [211] qui suit une approche similaire pour la gestion des données en s'appuyant en partie sur Globus. Enfin, Gridbus Broker [185] s'appuie également sur Globus pour implémenter et tester différentes politiques d'ordonnancement des données et des applications.

[1] Protocole faisant partie du système d'archivage sur bande Castor [49].
[2] Développé conjointement avec le projet DataGrid [77].
[3] Conjointement avec l'équipe Globus.

Discussion. L'ensemble des outils développés dans le cadre de cette approche offre un modèle d'accès *explicite* aux données. La gestion qui en découle reste donc à la charge du développeur des applications, qui s'exécutent sur les grilles. Cette gestion est donc un frein au développement à grande échelle de telles applications. Par ailleurs, ces systèmes permettent de stocker de manière persistante de grandes quantités de données, mais ne gèrent pas la cohérence des données partagées.

Toutefois et dans le cadre du projet DataGrid, un service de gestion de la cohérence (en anglais *Grid Consistency Service*, GCS [76]), appelé CONStanza [73], a été proposé. Ce projet vise une gestion de la cohérence relâchée des différentes copies d'une donnée, quelles soient stockées dans des bases de données ou dans des systèmes de fichiers. Cependant, le support annoncé des systèmes de fichiers n'est pas implémenté. Sa conception suit une approche à la Globus, à base de catalogues. La gestion des mises à jour s'appuie sur un protocole asynchrone, dans lequel les mises à jour sont périodiquement propagées depuis la copie de référence vers les copies secondaires. Ainsi, durant une certaine période, les données peuvent être incohérentes entre elles. Cet intervalle de temps définit le degré d'affaiblissement de la cohérence. Les auteurs se focalisent donc sur le cas où les écrivains ne peuvent modifier qu'une donnée de référence qui est fixe géographiquement dans le système. Enfin, les auteurs précisent que ce protocole n'est pas adéquat pour la gestion des métadonnées, sans toutefois apporter de solution.

3.2.2 Dépôts logistiques de données : l'approche IBP

Une deuxième approche pour la gestion des données s'inspire du concept de *réseau logistique* (en anglais *Logistical Network* [38, 37, 39]). Une telle approche vise à bâtir un réseau de stockage des données sur un ensemble de noeuds répartis dans différents endroits de l'infrastructure considérée. Les transferts de données entre deux machines transitent alors par un ou plusieurs de ces machines qualifiées d'intermédiaires, c'est-à-dire situés entre les sources et les destinations des transferts. Sur de tels réseaux, des techniques d'ordonnancement des transferts des données peuvent être mises en œuvre, parfois conjointement à l'ordonnancement des calculs.

Le projet IBP [143, 33] suit cette approche et constitue un exemple d'une telle infrastructure. Il se compare pour sa réalisation à la couche réseau TCP/IP. Son objectif est de concevoir une pile similaire mais pour la gestion à grande échelle de données utilisées par des systèmes ou des applications distribuées. Notons qu'IBP n'est qu'une couche de la pile proposée : il est souvent comparé à IP. En effet, IBP n'offre que des garanties limitées, du type « au mieux » (en anglais *best effort*). Ainsi, d'autres protocoles au-dessus d'IBP sont nécessaires, comme c'est le cas pour TCP au-dessus de IP. Nous décrivons dans la suite de cette section ces couches de bas en haut.

***Internet Backplane Protocol* (IBP).** L'entité de base dans IBP est un serveur de stockage, appelé *dépôt*. En théorie, un tel serveur de stockage peut être hébergé sur n'importe quel nœud de l'infrastructure matérielle. Toutefois en pratique il s'agit souvent d'une machine dédiée. L'interface de programmation d'IBP [35] permet d'allouer des tampons sur un ensemble de dépôts. Afin d'assurer un partage équitable des ressources, la politique d'allocation est basée à la fois sur le temps de stockage (noté $temps$) et la taille de l'espace de stockage (noté $espace$). Ainsi, un client ne peut allouer un ensemble de tampons de taille k, que si cette taille respecte la formule suivante : $k \leq espace * temps$.

Les primitives d'IBP permettent de lire et d'écrire depuis/dans ces tampons, mais également de réaliser des transferts entre dépôts (de 1 vers 1 ou de 1 vers n). Pour ce faire, IBP s'appuie sur le système de transferts de données *DataMover* [36, 159] qui peut utiliser différents protocoles (TCP, UDP, UDP multicast, etc.).

***Logistical Backbone* (L-Bone).** La couche L-Bone [34] permet à des clients d'utiliser un annuaire des différents dépôts IBP existants. Les métadonnées stockées dans ce répertoire sont les adresses TCP/IP des dépôts, leur quantité d'espace de stockage, leur localisation, leur performance, etc. L'objectif de L-Bone est de permettre à ces utilisateurs de sélectionner les dépôts IBP qui correspondent à leurs besoins.

***external Nodes* (exNodes).** Un exNode est à IBP ce qu'est un descripteur de fichier (en anglais *inode*) à un système de fichiers. Chaque appel pour allouer de l'espace de stockage dans IBP retourne une clé, également appelée une *capacité*. La connaissance de la valeur de cette clé est nécessaire pour accéder à la donnée. Un exNode est un regroupement d'un ensemble de capacités, ce qui permet par exemple de disposer d'un espace de stockage plus important, d'une disponibilité et de performances accrues grâce à la présence de plusieurs copies d'une même donnée dans différents dépôts IBP.

***Logistical Runtime System* (LoRS).** Cette couche vise à intégrer les couches L-Bone et exNodes dans une bibliothèque de haut niveau. L'objectif est de faciliter l'utilisation de ces couches, en automatisant une partie des étapes nécessaires pour, par exemple, créer un exNode et y insérer différentes capacités IBP.

La couche logicielle proposée par IBP ne vise pas particulièrement les grilles de calcul. Ainsi, son réseau publique est déployé sur le système PlanetLab [60]. Toutefois, l'utilisation du concept de réseau de logistique, et donc d'IBP, a été évaluée dans le contexte des grilles de calcul. Par exemple, IBP a été utilisé avec succès au sein de l'intergiciel Grid-RPC NetSolve pour la gestion des données [28], plus précisément l'ordonnancement conjointe du placement des données et des calculs. Dans ce contexte, l'utilisation d'IBP comme cache de données de matrices, pour des algorithmes de résolution de systèmes linéaires, a permis des améliorations de performances significatives.

L'approche choisie par IBP est également une *gestion explicite*, donc non transparente, des données. De manière similaire à Globus, un stockage persistant et un partage cohérent sont possibles, mais doivent être implémentés par les utilisateurs d'IBP. D'autres projets suivent une approche similaire à IBP. Nous en présentons brièvement quelques exemples ci-dessous.

Kangaroo [175] construit également un réseau de stockage en utilisant la hiérarchie des caches (mémoire et disque) des machines intermédiaires. L'objectif est de décharger l'application de l'envoi des données vers leur destination, ainsi que de la gestion des nombreuses erreurs qui peuvent se produire pendant ces transferts. Ainsi, la propagation des données produites par une application peut s'effectuer en parallèle avec la suite de son exécution. La cohérence des données n'est toutefois pas gérée.

NeST [40] permet à ses clients d'allouer des espaces de stockage, de manière similaire à IBP. Les données peuvent ainsi être transférées vers les systèmes de fichiers locaux des machines en ayant besoin pour réaliser des calculs. L'opération inverse est également réalisable. L'originalité de NeST réside dans son architecture à base de greffon (en anglais *plugin*). En effet, il s'agit d'un métaprotocole de transport qui peut utiliser de nombreux protocoles de transport : HTTP, FTP, GridFTP, NFS et Chirp. Par ailleurs, il peut également choisir dynamiquement le protocole de transport le plus adapté, ainsi

qu'optimiser ses paramètres (nombre de flux parallèles et/ou paramètres TCP). Une comparaison des performances de NeST et GridFTP est présentée dans [106]. Enfin, les applications peuvent utiliser de manière transparente NeST via un serveur NFS modifié.

DiskRouter [108] a pour objectif de construire un réseau de tampons, optimisant les débits des transferts des données produites par les processus applicatifs. Les auteurs estiment leur outil plus performant qu'IBP et Kangaroo, sans cependant en apporter la preuve. Pour atteindre son objectif de maximisation des débits, DiskRouter peut découper les flux de données pour les envoyer sur deux chemins différents. Par ailleurs, par rapport à GridFTP, DiskRouter optimise automatiquement ses paramètres de transferts, notamment son nombre de flux parallèles, la taille des tampons TCP, etc.

Ordonnancement. Ces différents systèmes suivent une approche à base de réseau logistique. Au-dessus de ceux-ci, d'autres systèmes peuvent mettre en place des politiques d'ordonnancement des données, parfois conjointement avec l'ordonnancement des calculs. Nous citons quelques systèmes ci-dessous.

[166, 167] [4] vise à améliorer le débit des applications qui s'exécutent sur les grilles de calcul. Pour ce faire, ce projet ordonnance les transferts des données en entrée des applications ainsi que les binaires eux-même des applications. Il s'appuie soit sur IBP soit sur un projet similaire appelé couche logistique de session (en anglais *Logistical Session Layer*, LSL [168]) pour la construction d'arbres de distribution de données, adaptés à l'état du réseau sous-jacent.

Stork [109] traite les données comme des tâches au même titre que les tâches d'exécution. Ce système définit plusieurs politiques d'ordonnancement pour les données. Pour ce faire, il utilise Condor ou Condor-G, qui ont été présentés à la section 2.3, et notamment son gestionnaire de dépendances DAGMan. Enfin, Stork est capable d'utiliser de nombreux protocoles de transports, tels que FTP, GridFTP, HTTP, DiskRouter et NeST. Par ailleurs, il est en mesure de sélectionner dynamiquement le meilleur protocole et d'optimiser ses paramètres [107]. Enfin, Stork peut utiliser les systèmes *Storage Resource Broker* (SRB) et *Storage Resource Middleware* (SRM) pour l'accès aux données. Ces deux systèmes sont présentés à la section 3.2.3.

Par ailleurs, le système NeST peut également implémenter des politiques d'ordonnancement variées pour les différents type de transferts. Cela est réalisable grâce à son architecture à base de greffon.

3.2.3 Système de fichiers : l'approche GFarm

Une troisième approche pour la gestion des données sur les grilles repose sur l'interface de programmation classique des système de fichiers. Ces systèmes visent à mettre en place sur les grilles de calcul des solutions similaires à NFS [153] qui a été conçu et utilisé sur les grappes de machines. Les systèmes qui suivent cette approche vont alors mettre en œuvre les mécanismes nécessaires pour localiser automatiquement une donnée lors de l'appel par un processus applicatif de la primitive `fopen` par exemple. L'objectif est d'éviter de modifier

[4] Les auteurs ne donnent pas de nom à leur projet.

les applications existantes en leur fournissant une interface de programmation par fichier. De nombreuses techniques existent pour atteindre cet objectif, telles que intercepter les appels à la bibliothèque C (en anglais *preloading*), modifier le noyau, etc. Ces techniques sont comparées dans [40]. Dans cette section, nous présentons un système de fichiers pour la grille basé sur la technique d'interception d'appels : GFarm.

GFarm [173] est un système de fichiers parallèles pour grilles, sur lequel s'exécute des applications nécessitant des pétaoctets de données. Dans GFarm, chaque nœud de calcul sert d'espace de stockage. Les données gérées par GFarm sont découpées et les fragments générés sont répartis sur différentes machines de calcul. GFarm suit alors la politique d'accès aux données qui consiste à placer les calculs sur les machines qui disposent des données nécessaires pour la réalisation de ces calculs. GFarm est donc adapté aux applications qui se présentent sous la forme de tâches indépendantes, chaque tâche possédant des propriétés de localité pour l'accès à un ensemble de données. Notons toutefois qu'après ce placement initial des processus, aucune migration des tâches vers les données n'est implémentée. GFarm vise principalement des données non-modifiables et les données peuvent être automatiquement répliquées d'un site à un autre lorsqu'elles sont accédées. Toutefois les métadonnées peuvent être modifiées. L'ouverture d'un fichier en écriture ainsi que l'acquisition de verrous sur des parties d'un fichier sont également possibles depuis GFarm version 2 [174]. Cette dernière version de ce système suit un modèle de cohérence à la NFS, par invalidation des copies d'un fichier lors de la fermeture d'une copie ouverte en écriture. Enfin, les expérimentations de GFarm présentent des performances prometteuses en termes de débit pour la lecture et l'écriture de données. Cela est principalement dû à l'interface parallèle de GFarm.

GridNFS [99] est un projet similaire à GFarm qui vise également à mettre en place un système de fichiers distribués pour la grille. Comme son nom l'indique, ce projet s'appuie sur la version 4 du protocole NFS [156]. Ce dernier protocole offre des nouvelles fonctionnalités par rapport à ses versions précédentes : sécurité accrue, support de la réplication en lecture, support de la migration, etc. GridNFS se sert également de Globus pour la gestion de la sécurité. L'objectif est la gestion de données de manière transparente et surtout sécurisée, grâce à l'espace de nommage global offert par NFS. GridNFS tire également parti du projet pNFS [95], qui permet d'utiliser avec les même performance les capacités des systèmes de fichiers parallèles comme PVFS2 [122].

LegionFS [190] est également un système de fichiers pour la grille. Comme son nom l'indique il s'appuie sur Legion et son modèle objet. Ainsi, il est facilement extensible, par exemple pour implémenter des mécanismes de gestion de la cohérence ou de réplication. Toutefois, dans l'implémentation décrite, de tels mécanismes sont absents, puisque les applications visées ne nécessitent pas une gestion de la cohérence. LegionFS offre des performances supérieures à NFS. Une version modifiée d'un serveur NFS utilisant l'interface de programmation de LegionFS est disponible. Elle offre les même garanties de cohérence et permet aux applications existantes de bénéficier de LegionFS de manière transparente, toutefois avec des performances moindres. Une interface pour des accès parallèles à une même donnée est également disponible.

Enfin, les systèmes d'exploitation pour grilles WebOS et 9grid (voir section 2.3) disposent d'un système de fichiers à cette échelle. Le système de fichiers de WebOS est appelé WebFS

et implémente un modèle de cohérence à la NFS. 9grid repose sur un système de gestion de versions, ce qui signifie que la résolution des conflits est à la charge des utilisateurs. Le système d'exploitation pour grilles Vigne [149] propose un service de gestion de données qui prend en compte la contrainte de volatilité des processus. Toutefois, ce service permet uniquement de partager des données durant l'exécution d'une application. Vigne propose par ailleurs un autre service dont l'objectif est de stocker de manière persistante des données.

D'autres projets visent à fournir aux applications une interface de programmation de type système de fichiers. Toutefois contrairement aux précédents, ces systèmes ne visent pas dès le début à être des systèmes de fichiers pour grilles. Généralement, ils mettent en place des mécanismes ad-hoc pour fournir la vision d'un espace de nommage global utilisable à travers une interface de programmation de type fichier. Nous en énumérons quelques-uns ci-dessous.

Globus Access to Secondary Storage (GASS [42]) est un système issu de Globus. Il s'apparente à un cache local sur une machine, pour une application s'exécutant sur cette même machine. Ainsi, une application peut de manière transparente accéder à une donnée distante. Toutefois, l'interface de programmation de GASS nécessite d'utiliser des fonctions spécifiques pour l'ouverture et la fermeture des fichiers, telles que gass_fopen et gass_fclose. Ces fonctions mettent en œuvre les transferts automatiques de données, vers le cache local de la machine, en s'appuyant sur FTP, HTTP, etc. GASS suit un modèle où toutes les données sont tout d'abord localement préchargées dans l'espace disque de la machine. Un modèle en mode flux de données (en anglais *streaming*) est également disponible avec toutefois des performances moindres. Dans ce dernier modèle tous les accès transitent par le réseau vers la machine qui dispose du fichier sur son disque.

Legion [189] fournit deux autres interfaces différentes pour la gestion des données, outre le projet LegionFS. La première interface est identique aux commandes Unix, telles que cp, ls, etc., les opérations étant préfixées par legion. La deuxième interface est comparable à celle fournie par GASS. Legion est donc similaire à GASS pour l'accès aux données. Les performances qui sont présentées sont similaires à celles de GASS, mais moindres que celles de FTP.

Grifi [213] et SRBfs [244] sont des systèmes de fichiers pour grilles, qui s'appuient pour leur mise en œuvre sur respectivement GridFTP et le système SRB. SRB est présenté à la section 3.2.4. Nous présentons de manière conjointe ces deux systèmes car ils suivent une même approche pour leur implémentation. En effet, tous deux reposent sur le projet FUSE [207]. FUSE permet de rediriger les appels du système de fichiers de l'espace noyau vers une autre bibliothèque qui s'exécute en espace utilisateur. Cette technique permet d'éviter de modifier le noyau, tout en étant transparente pour l'application. Elle est a été intégrée dans la version 2.6.14 du noyau Linux. FUSE n'est donc pas un système de gestion de données pour grilles. En revanche, il permet de mettre en œuvre des systèmes de fichiers de manière transparente pour les applications sur la grille. FUSE est également utilisé par GFarm pour offrir une solution alternative d'implémentation, nommée GFarmFS-FUSE, basée sur cette technique de redirection des appels du système de fichiers.

3.2.4 Accès unifié aux données : l'approche SRB

Une quatrième et dernière approche pour la gestion des données sur les grilles repose sur la volonté d'uniformiser l'accès à différents média de stockage. Ces média peuvent par exemple être des systèmes de fichiers, des bases de données relationnelles, des bases de données XML, des systèmes d'archivage (en anglais *Mass Storage System*), etc. Notons que l'interface de programmation offerte n'est pas celle d'un système de fichiers, bien qu'il soit possible d'en implémenter une au-dessus. SRB [146, 32] est l'exemple le plus représentatif de ce type de système. Notons également que ce projet vise plutôt les grilles de données. Toutefois et comme nous l'avons indiqué à la section 2.1, cette distinction entre grilles de calcul et de données s'estompe peu à peu [28].

Storage Resource Broker (SRB [146, 32]) est un intergiciel pour grilles qui fournit une interface de programmation uniforme sur un ensemble de ressources de stockage hétérogènes. Ainsi, SRB permet d'accéder à des données stockées dans des systèmes d'archivage, tels que *High Performance Storage System* (HPSS [214]), *ADSTAR Distributed Storage Manager* (ADSM [194], maintenant *Tivoli Storage Manager*) et *Data Migration Facility* (DMF [202]), des systèmes de fichiers tels que UFS, NTFS, HFS+ et des bases de données, telles que Oracle, DB2 et Sysbase. SRB offre différentes techniques pour la gestion conjointe des données et des calculs. Une des options est de précharger entièrement les données nécessaires aux calculs. D'autres possibilités existent pour l'accès aux données, comme l'utilisation d'une interface de type fichier (l'espace de nommage commence par `srb://` dans ce cas), ou alors l'interface spécifique fournie par SRB qui se base sur l'utilisation d'objets SRB. Enfin, il est possible de déplacer les applications vers les données nécessaires à leur fonctionnement.

Trois entités forment l'architecture de SRB :
- les *clients SRB* qui utilisent les interfaces de programmation de SRB ;
- les *serveurs SRB* qui abstraient les différents systèmes de stockage utilisés ;
- un catalogue de métadonnées, appelé *Metadata Catalog* (MCAT), qui implémente l'espace de nommage global.

Le catalogue de métadonnées est utilisé pour améliorer les performances d'accès aux données en choisissant les plus proches copies, ainsi que pour augmenter la disponibilité des données. SRB vise des applications qui n'ont pas besoin d'une gestion de la cohérence des données. Cependant, les auteurs indiquent sans le décrire le support de la gestion de la cohérence. Enfin, nous pouvons également regretter l'absence d'évaluations poussées. L'utilisation de SRB est mentionnée comme réussie dans plusieurs projets, notamment pour la visualisation d'images tridimensionnelles de création du système solaire [146]. Toutefois, les auteurs ne précisent pas sur quels critères cette évaluation se fonde et ne fournit pas de chiffres permettant de comparer les performances de ce système avec celles des autres.

SRB offre donc une transparence de localisation des données à travers différents systèmes de stockage répartis sur une grille. L'implémentation OGSA-DAI (*Data Access and Integration* [235])] de la spécification DAIS (*Data Access and Integration Services* [200]), publiée par le groupe de travail GGF du même nom, vise un but similaire. En effet, cette interface OGSA compatible avec Globus a pour objectif de spécifier une interface de programmation commune pour l'accès à des bases de données relationnelles, des bases de données XML et des systèmes de fichiers.

Dans un contexte légèrement différent, le projet SRM vise à unifier l'interface de programmation des systèmes d'archivage. Ceux-ci sont très souvent utilisés sur les grilles de données.

Storage Resource Manager (SRM [158, 245]) est la spécification d'une interface de programmation (Web service) qui fournit une interface uniforme au-dessus des systèmes d'archivage (en anglais *Mass Storage System*) tels que HPSS [214], Jamsine [94], Castor [49] etc. Le système SRM StoRM [74] permet également l'accès à des systèmes de fichiers parallèles tels que GPFS et XFS. L'objectif de la spécification SRM est d'affranchir les couches supérieures, tel qu'un catalogue de réplicas par exemple, des particularités spécifiques à chaque système et donc de rendre interopérable les applications avec l'ensemble des systèmes implémentant la spécification SRM. La gestion des données peut s'effectuer de manière hiérarchique en ajoutant des disques en guise de cache des données issues de ces systèmes. L'espace de stockage nécessaire pour les applications peut être alloué par des commandes de l'interface. Différentes politiques de gestion des caches peuvent être mises en œuvre, en tenant compte des besoins des divers clients. Pour les transferts de données, les systèmes SRM s'appuient notamment sur GridFTP. Des exemples de SRM sont Fermilab [141][5] et StoRM [74]. Ils implémentent tous les deux l'interface SRM 2.1.

De manière très réductrice, SRB peut être vu comme un système offrant des fonctionnalités similaires à l'interface SRM, mais implémenté de manière distribuée. Notons qu'un projet a aussi implémenté une interface SRM au-dessus de SRB [78].

3.3 Critères d'évaluation et limites des systèmes présentés

Dans cette section, nous présentons les limites des différents systèmes que nous avons décrits tout au long de la section 3.2. Afin de comparer précisément ces systèmes, nous définissons tout d'abord des critères d'évaluation.

Transparence. Il s'agit d'évaluer la transparence de la localisation des données pour les utilisateurs. Par ailleurs, à chaque fois nous précisons le type d'interface de programmation offert par l'intergiciel de gestion de données ainsi que sa facilité d'utilisation, c'est-à-dire s'il offre des interfaces traditionnelles. Enfin, nous indiquons également si l'intergiciel s'occupe des données avant, pendant et/ou après l'exécution de l'application.

Cohérence. Il s'agit d'évaluer la transparence du partage de données mis en œuvre pour les utilisateurs. Par ailleurs, nous indiquons également quelles sont les garanties offertes par les systèmes, c'est-à-dire les modèles de cohérence utilisés. Enfin, nous précisons si les protocoles implémentant les modèles tiennent compte des caractéristiques physiques des grilles de calcul.

Persistance. Il s'agit d'évaluer la persistance du stockage des données, c'est-à-dire garantir que le temps de stockage soit supérieur à l'exécution d'une ou plusieurs applications qui ont besoin de ces données. Nous indiquons à chaque fois sur quel type de support (mémoire vive, disque, bande, etc.) s'effectue le stockage.

[5] Composé du cache disque distribué dCache et du système d'archivage sur bande Enstore.

Tolérance à la volatilité. Il s'agit d'évaluer la tolérance à la volatilité des entités qui composent l'intergiciel de gestion des données (voir définition 2.6), vis-à-vis de l'infrastructure physique sous-jacente. Le support de cette volatilité est une condition nécessaire pour le passage à l'échelle de l'intergiciel de gestion de données.

Performance. Il s'agit d'évaluer la performance de l'intergiciel de gestion de données, par exemple, le débit d'une lecture/écriture, son débit agrégé, le temps de latence pour accéder à une donnée, etc.

Le tableau 3.1 récapitule les critères auxquels satisfont les différents travaux apparentés pour la gestion des données sur les grilles de calcul. Les différents protocoles de transports détaillés à la section 3.2.1 ne sont pas représentés en raison de leur caractère trop bas niveau. Il ne forment pas individuellement un système de gestion de données, mais sont indispensables pour leur construction. De plus, notons qu'ils ne visent principalement qu'au transport des données avant et après l'exécution d'une application. Ce tableau permet de constater qu'aucun des systèmes étudiés[6] ne satisfait *simultanément* aux propriétés de *tolérance à la volatilité* et de *partage cohérent* des données. D'ailleurs peu de systèmes satisfont même seulement à l'une ou l'autre de ces deux propriétés. Par exemple, la propriété de tolérance à la volatilité n'est que peu souvent abordée dans les articles décrivant ces systèmes. Elle est cependant prise en compte, mais partiellement, par quelques systèmes qui suivent une approche à base de dépôts logistiques de données : DiskRouter et Stork. Concernant le partage cohérent, l'approche à base de systèmes de fichiers traite pour certains de ces systèmes de cette propriété. Ainsi, GFarm, LegionFS, GridNFS, WebFS et 9grid implémentent au moins un modèle de cohérence. Toutefois, il s'agit généralement du modèle de cohérence de NFS[7] qui n'offre pas des garanties satisfaisantes pour les applications scientifiques que nous visons. Seuls les systèmes CONStanza et le service de gestion de données volatiles du système Vigne visent une gestion des données où celles-ci peuvent être partagées de manière cohérente, entre les différents processus d'une application. Le service de gestion de données volatiles du système Vigne présente l'avantage d'offrir une transparence de la localisation des données, ainsi qu'une tolérance à la volatilité. Toutefois, ce service ne permet pas un stockage persistant des données et donc un partage de celles-ci entre plusieurs applications. Enfin, son évaluation a été réalisée à l'échelle d'une grappe de machines et non d'une grille de calcul. De son côté CONSTanza a principalement évalué en utilisant une implémentation fondée sur des bases de données, et son mécanisme de propagation des mises à jour est coûteux. De plus, ce système n'offre pas une transparence de la localisation des données.

3.4 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté la problématique de la gestion des données dans les grilles de calculs. Puis nous avons décrit quatre projets différents qui abordent la problématique précédente : l'ensemble des services Globus qui s'occupent de la gestion des données, la couche logicielle basée sur IBP, le système de fichiers GFarm et enfin l'intergiciel SRB issu des grilles de données. À travers ces quatre projets, nous avons présenté un panorama des solutions pour la gestion des données dans les grilles de calcul.

[6] Excepté le service de gestion de données volatiles du système Vigne.
[7] Exception faite de 9grid.

	Transparence	Cohérence	Persistance	Tolérance à la volatilité	Performance
Globus	×	×	√	×	√
CONStanza	×	√	√	×	□
IBP	×	×	×	×	√
LoRS	×	×	√	□	√
Kangaroo	×	×	×	×	√
NeST	×	×	×	×	√
DiskRouter	×	×	×	∼	√
[167]	×	×	×	×	√
Stork	×	×	×	∼	√
GFarm	√	∼	√	□	√
LegionFS	√	∼	√	□	√
GridNFS	√	∼	√	□	□
GASS	∼	×	×	□	×
Legion	√	×	√	□	×
Grifi	√	×	√	□	□
WebFS	√	∼	√	□	□
9grid	√	∼	√	□	□
Vigne	√	√	×	√	□
SRB	√	□	√	□	∼
OGSA-DAI	×	×	√	□	□
SRM	×	×	√	□	□

TAB. 3.1 – Récapitulatif des propriétés des travaux apparentés à la gestion de données dans les grilles de calcul. √ : propriété satisfaite ; × : propriété non satisfaite ; ∼ : propriété partiellement satisfaite. □ : propriété non évoquée.

Ensuite, nous avons défini nos critères d'évaluation de ces projets aux besoins des applications que nous visons. Ainsi, nous avons pu constater qu'aucun système étudié[6] ne satisfait de manière simultanée à la propriété de tolérance à la volatilité et au partage cohérent des données. Cette absence de tolérance à la volatilité s'explique principalement par l'échelle des grilles utilisée. En effet, tous les systèmes que nous avons présentés ont été développés sur des grilles de taille relativement faible, comme le montre les expériences relatées dans les articles décrivant ces systèmes. Toutefois, compte tenu de la croissance actuelle de l'échelle des grilles, cette contrainte de volatilité prend de plus en plus d'importance. L'absence de gestion cohérente des données est due aux besoins des applications jusque-là visés par ces systèmes. En effet, les applications scientifiques gèrent de grandes quantités de données mais généralement en lecture. Notons cependant que les applications de couplage de code nécessitent un partage de données cohérent par construction. Par ailleurs, les métadonnées des données générées par les expériences de physique, par exemple, nécessitent une gestion de la cohérence. Enfin, ce besoin grandit comme le prouve l'existence du système CONStanza et du service de gestion de données volatiles du système Vigne pour partager des données de manière cohérente.

Avant d'introduire dans la deuxième partie de ce manuscrit notre contribution à la gestion des données sur les grilles, nous présentons dans le chapitre suivant deux types de systèmes distribués. Il s'agit des systèmes à *mémoire virtuellement partagée* (MVP) et des systèmes *pair-à-pair* (P2P). Chaque type de système a indépendamment rempli la propriété de tolérance à la volatilité ou de partage cohérent des données. Ils constituent donc des sources d'inspiration pour la gestion des données sur les grilles.

Chapitre 4

Deux approches pour la gestion des données dans les systèmes distribués

Sommaire

Dans ce dernier chapitre de cette première partie, nous présentons deux approches pour la gestion des données dans les systèmes distribués. Nous introduisons tout d'abord les systèmes à mémoire virtuellement partagée (MVP) (en anglais *Distributed Shared Memory*), puis nous décrivons les systèmes pair-à-pair (P2P), notamment trois dédiés au partage de fichiers. Dans les deux cas, notre présentation est la suivante : nous définissons tout d'abord ces types de systèmes, puis nous décrivons leurs principes de fonctionnement. Ensuite, nous en montrons les limites et nous terminons par relater quelques tentatives ayant pour but de corriger ces insuffisances.

L'objectif de ces présentations est d'étudier deux types de systèmes dotés chacun d'une propriété qui fait défaut aux systèmes de gestion de données sur les grilles. Les systèmes P2P ont la propriété de tolérance à la volatilité, alors que les systèmes à MVP permettent de partager de manière cohérente des données répliquées et modifiables. Ainsi, ces systèmes peuvent être une source d'inspiration intéressante pour la réalisation d'un service de partage

de données pour grille qui intègre ces deux propriétés. En conséquence, notre description de ces deux types de systèmes n'est pas une étude exhaustive, mais vise plutôt à souligner ces propriétés.

4.1 Les systèmes à mémoire virtuellement partagée

Dans cette section, nous définissons et présentons le principe fonctionnement des systèmes à mémoire virtuellement partagée (MVP). Puis, nous présentons les limites de ces systèmes et quelques tentatives pour remédier à ces déficiences.

4.1.1 Définition et principe de fonctionnement

L'idée d'utiliser des systèmes à MVP[1] a été proposée par Kai Li en 1986 [117]. Définissons tout d'abord la notion de système à MVP.

Définition 4.1 : ***système à* MVP** — Mémoire virtuelle globale, partagée par plusieurs processus, au-dessus d'un ensemble de mémoires physiques distribuées sur différentes machines interconnectées par des équipements réseaux.

Notons qu'un système à MVP (logiciel uniquement) peut être considéré comme un intergiciel, au sens de la définition 2.8. Le concept de système à MVP vise à offrir aux développeurs d'applications le modèle de programmation des machines multiprocesseurs à mémoire partagée au-dessus de machines multiprocesseurs à mémoire distribuée. Le rôle d'un système à MVP est alors de : 1) projeter des régions de mémoire physique distinctes dans une seule et même mémoire virtuelle, 2) localiser les données et y accéder et 3) préserver la cohérence de l'ensemble de l'espace mémoire par rapport aux différents processeurs. Les programmeurs peuvent ainsi partager des données de manière *transparente*, au lieu d'utiliser des systèmes à base de passage de messages, qui nécessitent de calculer quelles données partager et quand les transférer.

En effet, lorsqu'une application est exécutée au-dessus d'un système à MVP, la localisation et la cohérence des données partagées en mémoire sont automatiquement prises en charge. Cela est possible même en présence de plusieurs processus applicatifs ou processus légers distribués qui accèdent aux mêmes données partagées. En conséquence, les programmeurs peuvent se focaliser sur le code métier de leurs applications et non sur des problèmes de bas niveau, tel que l'échange de données. La complexité de programmation de systèmes distribués est donc réduite par les systèmes à MVP. Toutefois, cette transparence a un coût : les performances des systèmes à MVP sont en effet moindres, par rapport à des programmes bâtis sur une bibliothèque de passage de messages, telle que MPI. En effet, dans ce dernier type de programmes, les échanges explicites de données peuvent être finement optimisés.

Les systèmes à MVP peuvent se classifier par leurs modèles et leurs protocoles de cohérence, mais également par leurs choix d'implémentation. Les modèles et les protocoles de cohérence font l'objet de la section 4.1.2. Nous décrivons brièvement dans la suite de cette section les différents choix possibles d'implémentation d'un système à MVP.

[1] Logiciels et non matériels. Cette distinction est précisée dans la suite de cette section.

Niveau d'implémentation. Il définit le niveau auquel se situe un système à MVP, par rapport au matériel, au système d'exploitation, etc. On distingue principalement trois niveaux d'implémentation pour les systèmes à MVP : *matériel*, *logiciel* et *mixte*. L'avantage des systèmes à MVP matériels est d'offrir une transparence complète pour l'utilisateur, une granularité de partage fine et de très bonnes performances. Un exemple de tel type de système à MVP sont les machines SGI Origin2000. Leur désavantage est un coût important. Les systèmes à MVP logiciels sont généralement implémentés sous la forme d'une bibliothèque et/ou de modifications dans les compilateurs et/ou de modifications du système d'exploitation. Dans les deux premiers cas, on parle de systèmes à MVP logiciels implémentés au niveau utilisateur, et dans le dernier cas d'une implémentation au niveau noyau. La flexibilité est le grand avantage des systèmes à MVP logiciels, malgré des performances généralement moindres que dans le cas des systèmes à MVP matériels. Enfin, des systèmes à MVP hybrides reposent à la fois sur des mécanismes matériels spécifiques, mais également sur des couches logicielles pour leur implémentation. Dans la suite de cette section, nous ne faisons plus cette distinction entre les systèmes à MVP.

Granularité de partage. Elle définit la taille unitaire des blocs de données manipulés par un système à MVP. Elle varie de la taille d'une ligne de cache à la taille d'une page mémoire. Le premier cas se retrouve principalement dans les systèmes à MVP matériels, la granularité étant alors de l'ordre de quelques octets. Le dernier cas est typique des systèmes à MVP logiciels. En effet, ceux-ci utilisent le mécanisme classique de défaut de page. La taille de la granularité est alors généralement la taille d'une page mémoire du système d'exploitation, par exemple 4 ko ou 8 ko. De manière générale, une granularité importante permet de profiter de la localité spatiale des accès à la mémoire réalisés par les applications, par préchargement des données contenues dans une page. Toutefois, le problème dit du *faux-partage* peut alors apparaître. Il consiste en l'accès alterné à des données distinctes situées dans la même page mémoire, par deux processus différents non co-localisés sur une même machine. La page mémoire est alors continuellement transférée entre les deux processus, ce qui engendre des pertes de performance significatives.

Gestion de la localisation. Avant de pouvoir autoriser à un processus des accès mémoire à une donnée, la page mémoire correspondante doit être localisée par le système à MVP. En effet, une page mémoire peut être distante par rapport au processus qui souhaite y accéder. La localisation de la machine qui détient la copie à jour de cette page mémoire, appelé propriétaire, peut être réalisée de plusieurs manières. On appelle gestionnaire d'une donnée le nœud qui s'occupe de localiser le dernier propriétaire dans le système. Les gestionnaires peuvent être *centralisés* ou *distribués* et dans le dernier cas *statiques* ou *dynamiques*. Le protocole de cohérence présenté dans [119] repose sur une mise en œuvre à base de gestionnaires distribués dynamiques. Dans cet algorithme, chaque nœud du système stocke le propriétaire probable de la page. Cette information permet aux requêtes d'aboutir au véritable propriétaire de la page mémoire en une complexité logarithmique en fonction du nombre de processeurs. Une autre approche a été proposée : elle consiste à fusionner le rôle de gestionnaire et de propriétaire. Dans une telle approche, appelée nœud-hôte (en anglais *home-based*), un nœud est statiquement à la fois gestionnaire et propriétaire d'un ensemble de pages mémoire.

Stratégies algorithmiques. La mise en œuvre d'un système à MVP peut s'appuyer sur dif-

férents choix, ou stratégies, algorithmiques. Ainsi, outre la localisation des propriétaires, un système à MVP doit choisir entre migrer ou répliquer une page mémoire accédée. La migration consiste à ne disposer que d'un unique exemplaire de la donnée, qui est déplacé selon les accès. En revanche, la réplication autorise l'existence de plusieurs copies de la donnée dans le système. L'avantage est alors d'augmenter les performances des applications parallèles en favorisant la localité physique des pages mémoire par rapport aux machines qui les utilisent. Toutefois, la réplication nécessite de gérer la cohérence de l'ensemble des copies.

Une autre stratégie algorithmique à laquelle un système à MVP doit faire face est le nombre d'écrivains/lecteurs autorisés à écrire/lire, de manière simultanée, des données situées dans une même zone. La taille de cette zone correspond à la granularité de partage, typiquement une page mémoire. Les modèles les plus intéressants sont : 1) lecteurs multiples et écrivain unique, en anglais *Multiple Readers, Single Writer* (MRSW) et 2) lecteurs et écrivains multiples, en anglais *Multiple Readers, Multiple Writers* (MRMW). Dans la première possibilité, les données sont répliquées dans le système, afin d'autoriser plusieurs lecteurs à lire en parallèle la même donnée. Les différentes copies de la donnée seront invalidées par le protocole de cohérence lors de toute modification par un écrivain. Dans la seconde possibilité, plusieurs écrivains sont autorisés à modifier une même donnée d'une page mémoire mais à différents endroits, ou plusieurs données différentes stockées sur la même page mémoire. Pour ce faire, le modèle MRMW s'appuie généralement sur des techniques de différence du contenu des pages (en anglais *diffing*) et de copie jumelle (en anglais *twin*) pour la propagation des mises à jour d'une donnée.

4.1.2 Modèles et protocoles de cohérence

Afin d'offrir cette vision d'une mémoire virtuelle globale et d'un partage cohérent des données dans cet espace, les systèmes à MVP ont développé des modèles et des protocoles de cohérence. Définissons tout d'abord ce que représente un modèle de cohérence.

Définition 4.2 : ***modèle de cohérence*** — Ensemble de règles qui spécifie l'ordre dans lequel les accès à des données partagées par un processeur sont visibles par l'ensemble des processeurs du système.

Le respect par le programmeur d'un ensemble de contraintes de programmation imposées par le modèle de cohérence est une condition nécessaire et suffisante pour obtenir les garanties offertes par le modèle utilisé. Ainsi, un modèle de cohérence est un « contrat » passé entre le système à MVP et le programmeur au sujet de la gestion de la cohérence.

Définition 4.3 : ***protocole de cohérence*** — Algorithme de gestion des données respectant les contraintes et les garanties d'un modèle de cohérence.

Notons que plusieurs protocoles de cohérence peuvent implémenter un même modèle de cohérence.

Il existe différents modèles de cohérence. Par exemple, le modèle de la cohérence séquentielle [115] est un modèle dans lequel les accès mémoire sont vus dans le même ordre total sur l'ensemble des processus d'une application. Il est par exemple utilisé dans le système

à MVP IVY [118]. Ce modèle fait partie de la catégorie des *modèles de cohérence forte*. Dans cette catégorie de modèle, tous les accès à la mémoire se font sur des données cohérentes. Le choix d'un modèle de cohérence détermine les performances des systèmes à MVP. En effet, assurer une cohérence forte a un coût important en termes de communication. C'est pour cette raison que la recherche réalisée dans les systèmes à MVP s'est concentrée sur l'élaboration de modèles de cohérence plus relâchés par rapport aux modèles de cohérence forte. On parle alors de *modèle de cohérence faible*. Dans ces modèles, les accès aux données partagées sont distingués des appels sur les primitives de synchronisation des données. Les données qui sont comprises dans une section délimitée par ces primitives de synchronisation, également appelée *section critique*, sont alors soumises à un modèle de cohérence forte, généralement la cohérence séquentielle. Cette distinction permet d'éviter de propager inutilement des modifications à d'autres machines, et ainsi d'augmenter ainsi les performances des systèmes à MVP. En effet, la cohérence des données n'est assurée que lors des appels aux primitives de synchronisation et non à chaque accès mémoire comme c'est le cas dans les modèles de cohérence forte. Un certain nombre de ces modèles de cohérence faible sont présentés ci-dessous.

Cohérence faible (en anglais *weak consistency*). Modèle de cohérence qui le premier introduit l'utilisation de *variables de synchronisation* pour délimiter des sections critiques. À la fin d'une section critique, toutes les modifications locales sont propagées vers les autres machines du système à MVP, et inversement la mémoire locale prend en compte les modifications effectuées par les autres machines. Dans une même section critique, ce modèle évite donc de propager inutilement les modifications apportées aux données à chaque accès en écriture. La mémoire est cohérente uniquement lors des accès de synchronisation.

Cohérence à la libération (en anglais *release consistency*). Contrairement au modèle précédent, ce modèle de cohérence distingue l'entrée d'une section critique de sa sortie. L'objectif est de réduire les opérations réseau nécessaires au maintien de la cohérence, en n'effectuant que les actions nécessaires pour l'entrée ou la sortie d'une section critique. Deux classes de protocole existent pour implémenter ce modèle. Une possibilité est de mettre à jour sur les autres machines les données partagées d'une section critique lors de relâchement du verrou protégeant cette section. On parle de *cohérence à la libération immédiate* (en anglais *eager release consistency*). Toutefois, ces modifications ne sont nécessaires par d'autres machines que lors d'une prochaine entrée en section critique. Elles peuvent donc simplement être propagées à ce moment. On parle alors de *cohérence à la libération paresseuse* (en anglais *lazy release consistency*). La cohérence à la libération immédiate est mise en œuvre dans le système à MVP Munin [55], et la cohérence à la libération paresseuse dans le système à MVP TreadMarks [20].

Cohérence d'entrée (en anglais *entry consistency*). Modèle de cohérence à la libération où chaque variable partagée est protégée par un verrou spécifique à la donnée. Ainsi, lors de l'acquisition d'un verrou, seule la donnée associée à ce verrou est mise à jour. L'objectif de ce modèle de cohérence est de réduire les problèmes de faux partage qu'engendre le modèle de cohérence à la libération. Les accès aux verrous se font en mode *exclusif* ou *partagé*. Dans ce dernier cas, plusieurs processus peuvent acquérir un même verrou pour lire simultanément la même donnée. Le système à MVP Midway [41] implémente ce modèle.

Cohérence de portée (en anglais *scope consistency*). Modèle de cohérence qui détermine

implicitement les données à mettre à jour lors d'une entrée en section critique [101]. L'objectif de ce modèle est d'épargner ce travail aux développeurs d'applications. Un exemple de système à MVP qui implémente un tel modèle de cohérence est Brazos [161].

Un système à MVP n'implémente que quelques modèles de cohérence, généralement un ou deux. Ces systèmes sont généralement difficilement extensibles à de nouveaux protocoles de cohérence. Dans ce contexte, soulignons les approches qui visent à fournir un système à MVP générique, c'est-à-dire permettant facilement de mettre en œuvre différents protocoles de cohérence. Un exemple d'un tel système à MVP est DSM-PM^2 [22].

4.1.3 Les limites

Les systèmes à MVP ont l'avantage d'offrir un modèle de programmation simple aux développeurs d'applications distribuées. Toutefois, ces systèmes s'exécutent généralement sur des grappes de machines, c'est-à-dire à *petite échelle*, de l'ordre de quelques dizaines de machines. Par ailleurs, les systèmes à MVP supposent implicitement une *architecture homogène*, en termes de processeur, de système d'exploitation, etc. Enfin, les systèmes à MVP ne tolèrent généralement pas les fautes, dues par exemple à des conditions de volatilité. Plusieurs systèmes ont été proposés afin de remédier à ces défauts, qui deviennent des facteurs limitants dans le contexte d'une utilisation sur les grilles de calcul. Nous en présentons trois dans cette section.

Teamster-G [120] est un système à MVP qui vise les grilles de calcul. Il permet de construire une MVP sur une grappe virtuelle de machines, bien que les différentes machines qui la composent soient physiquement distribuées dans plusieurs sites d'une grille. Ce système doit donc prendre en compte les caractéristiques des grilles de calcul (voir chapitre 2), qui sont bien différentes de celles des grappes de machines. Pour sa mise en œuvre, Teamster-G s'appuie sur le système à MVP Teamster et Globus 3. Ce dernier est utilisé pour l'allocation des ressources, la gestion de la sécurité, le préchargement des données et des binaires (via GASS) ainsi que la surveillance des ressources d'une grille. La principale contribution annoncée porte sur l'allocation de ressources. Un protocole à session a été développé pour la construction de la MVP, notamment sur plusieurs grappes de machines. Toutefois, aucune explication n'est donnée sur son algorithmique dans ce contexte multigrappes. Les auteurs présentent cependant des exécutions d'applications sur deux grappes de machines. Des pertes de performances, faibles ou fortes, selon le type d'applications utilisées, sont observées par rapport à une exécution sur une unique grappe de machines. Par ailleurs, un protocole de cohérence hiérarchique implémentant des caches de données par grappe des machines a été développé pour minimiser le nombre de messages en WAN. Enfin, notons que Teamster-G ne gère pas la tolérance aux fautes et que le sujet de l'hétérogénéité (petit-boutiste ou grand-boutiste) n'est également pas abordé.

Mome [102] est un système à MVP qui vise notamment à faciliter le partage de données dans les applications parallèles et de couplages de code s'exécutant sur les grilles de calcul. Mome fournit un espace de segments partagés à des programmes qui s'exécutent sur des calculateurs à mémoire partagée ou des grappes de machines. Un processus dans Mome peut demander la projection des segments partagés dans son espace

d'adressage (partiellement ou totalement). Les segments mémoire de Mome offrent un espace d'adressage où tous les objets partagés de l'application parallèle sont alloués. Ainsi, les bibliothèques de couplage de code n'ont pas à se soucier de la localisation des données, ce qui est un bénéfice important. Mome gère la cohérence des copies des segments avec la granularité des pages virtuelles du système d'exploitation sous-jacent. Chaque processus peut choisir le modèle de cohérence qui doit être appliqué à sa propre vue de chaque page partagée. Deux modèles de cohérence sont disponibles : le modèle de cohérence stricte et le modèle de cohérence relâchée. Toutefois, Mome a seulement été évalué sur quelques machines, même si celles-ci étaient situées dans deux grappes de machines différentes. Par ailleurs, Mome a été validé par intégration dans un système OpenMP. Enfin, notons que les problématiques de tolérance aux fautes et de gestion de l'hétérogénéité ne sont pas abordées dans les articles décrivant Mome.

InterWeave [57] est également un système à MVP à base de segments mémoire pour l'exécution d'applications distribuées à grande échelle. Il est similaire à Mome. Il permet aux processus applicatifs de spécifier des zones, appelées vues. Seules les données présentes de ces vues sont alors soumises au modèle de cohérence choisi. L'objectif est d'améliorer les performances du système. InterWeave a également défini de nouveaux modèles de cohérence, tel que le modèle de cohérence à base de différences (en anglais *delta coherence*). Ce modèle garantit que le segment mémoire n'a pas plus de x versions de retard par rapport à la copie de référence.

Notons que d'autres systèmes à MVP ont été proposés, notamment pour la tolérance aux fautes, tel que dans [105] par exemple. Par ailleurs, l'implémentation des modèles de cohérence a fait l'objet de recherches, notamment afin de tenir compte des différences de latence dans des infrastructures *hiérarchiques* composées de plusieurs grappes de machines. Par exemple, le protocole CLRC [24] propose une implémentation adaptée au contexte multigrappe, de la cohérence à la libération paresseuse. Enfin, dans [23] le protocole de synchronisation des accès mémoire a également été adapté à la hiérarchie des latences présentes dans les infrastructures composées de plusieurs grappes de machines.

4.2 Les systèmes de partage de données pair-à-pair

Dans cette section, nous présentons les systèmes de partage de données pair-à-pair (P2P), en suivant le même plan qu'à la section précédente.

4.2.1 Définition et principe de fonctionnement

Internet est majoritairement composé de machines connectées par intermittence, qui constituent en quelque sorte sa *périphérie* (par opposition aux différents types de serveurs ayant une présence permanente et qui forment son cœur). Ces machines disposent de capacités diverses non utilisées : puissance de calcul, espace disque, etc. L'apparition du *modèle pair-à-pair* (en anglais *peer-to-peer*, P2P [123, 157]) a été motivée par le souhait de faire collaborer cet ensemble de ressources non utilisées dans le modèle actuel d'Internet. Ce modèle complémente donc le modèle classique *client-serveur*. Dans le modèle P2P, chaque processus applicatif peut être à la fois client dans une transaction et serveur dans une autre. L'objectif

initial de ce modèle est la réduction des coûts par agrégation des ressources disponibles sur Internet.

Le modèle P2P a récemment attiré l'intérêt de la communauté scientifique, de par ses propriétés intéressantes. Un système est dit être P2P lorsqu'il suit le modèle de programmation P2P. Nous adoptons la définition suivante, issue en partie de [193], pour un système P2P :

Définition 4.4 : *système pair-à-pair (P2P)* — Système distribué à grande échelle, auto-adaptatif et décentralisé, dans lequel 1) toutes ou la majeure partie des communications entre les processus sont symétriques et 2) chaque processus peut potentiellement exécuter l'ensemble des rôles définis par le système.

Un système P2P est implémenté soit par l'utilisation d'un intergiciel, au sens de la définition 2.8, soit directement au sein de l'application. Dans le dernier cas, on parle alors de système P2P s'appuyant pour sa mise en œuvre sur des protocoles spécifiques, voire propriétaires. Des exemples d'intergiciels P2P sont JXTA-C [179] et FreePastry [206]. Un exemple d'application P2P utilisant un protocole spécifique est donné par le logiciel de partage de données KazaA [224].

Un processus d'un système P2P est généralement appelé un *pair*. Un pair est une entité qui peut offrir des ressources comme sa capacité de calcul ou son espace de stockage et des services. Un réseau P2P ou réseau logique (en anglais *overlay*) définit comment les pairs d'un système P2P sont connectés entre eux. Un pair A est dit être connecté à un pair B si l'identifiant du pair B est présent dans la table de routage du pair A. Notons que la connexion au niveau réseau P2P n'implique pas une connexion directe entre les deux pairs, au niveau TCP.

Les deux principales propriétés qui caractérisent un système P2P sont la tolérance à la volatilité et l'extensibilité.

La tolérance à la volatilité des entités constituant un système P2P permet de prendre en compte la dynamicité du réseau afin de supporter l'ajout et le retrait de pairs dans le système. En effet, un système P2P compte tenu de sa grande échelle est sujet aux modifications dans la configuration des pairs qui forment le système. Cette dynamicité concerne également les capacités des pairs, suite par exemple aux fluctuations des débits des connexions reliant les pairs.

L'extensibilité permet de supporter un grand nombre de pairs, de l'ordre de plusieurs millions. Elle passe par la décentralisation de la logique de contrôle du système à l'ensemble des pairs, afin d'éviter des points critiques sujets aux fautes. Cette propriété est liée à la précédente : une tolérance à la volatilité permet d'envisager une meilleure extensibilité du système.

Popularisé par Napster [232], Gnutella [138], et aujourd'hui KaZaA [224], le paradigme P2P a été centré dès le départ sur la *gestion de larges masses de données réparties à très grande échelle*. Il a montré qu'il offre une véritable solution dans ce contexte. À titre d'exemple, le réseau KaZaA [224], centré sur le partage de fichiers à grande échelle, regroupe en moyenne, à chaque instant, 4.500.000 machines, 900.000.000 fichiers contenant 9 pétaoctets de données. Toutefois, les objectifs des systèmes P2P sont variés. Certains visent à garantir la persistance des données alors que d'autres s'attachent à diffuser ces mêmes données sans en assurer la

pérennité. Certains privilégient l'anonymat au détriment de l'efficacité, d'autres visent tout simplement des services spécifiques comme la téléphonie sur Internet.

Dans la suite de cette section, nous nous focalisons uniquement sur les systèmes de partage de données P2P. Les principaux objectifs de ces systèmes sont les suivants :

Tolérance aux fautes par la réplication des données. L'objectif est de garantir, lors du retrait de pairs, un certain niveau de persistance des données.

Disponibilité des données par l'utilisation de mécanismes de caches. Les systèmes P2P souhaitent minimiser le temps de récupération des données pour un débit fixé. L'objectif consiste donc à placer une donnée le plus près possible d'un pair qui va en faire la demande.

L'implémentation d'un système P2P de gestion de données repose généralement sur deux couches : une *couche de stockage* qui s'appuie elle-même sur une *couche de routage*. Une description détaillée du rôle de chaque couche est donnée dans [65]. Notons qu'un système P2P de gestion de données peut fournir une interface de type système de fichiers.

4.2.2 Localisation des données et routage

Le principal rôle d'un système P2P est d'acheminer les requêtes et les réponses de ses différents processus P2P, via sa couche de routage. Il existe plusieurs approches pour l'implémentation de cette couche de routage. Ces approches permettent de classifier les systèmes P2P selon leur type d'architecture, également appelé *topologie*.

Par répertoire centralisé. La localisation repose sur un serveur central qui regroupe, dans un répertoire, l'ensemble des couples (pair, ressource) des ressources présentes sur l'ensemble des pairs formant le système. Napster en est le parfait exemple. Lorsqu'une donnée a été localisée dans le système par un pair, c'est-à-dire qu'il a récupéré du serveur central l'adresse d'un autre pair qui héberge cette donnée, le routage se fait alors directement entre ces deux pairs. Toutefois, la défaillance du serveur central suffit à entraîner l'indisponibilité du service rendu par l'application basée sur cette technique.

Par inondation. La localisation repose sur la diffusion de la requête (en anglais *broadcast*) à l'ensemble des pairs qui sont à l'intérieur d'un certain voisinage. Ce voisinage est l'ensemble des pairs du système qu'un pair donné connaît, et définit l'horizon que le pair peut atteindre. Gnutella utilise cette technique de routage. Des optimisations sont possibles afin de limiter la diffusion à un sous-ensemble de pairs, en utilisant par exemple des filtres de Bloom. Une variante utilise la notion de *super-pair*. Un super-pair est un pair qui agit comme un répertoire centralisé pour le compte d'un ensemble restreint de pairs, généralement « proches ». La propagation des requêtes se fait alors par diffusion entre ces super-pairs. KaZaA est l'un des systèmes qui se basent sur cette technique. Dans les deux approches, le routage vers le pair qui a émis la requête se fait alors, depuis un pair hébergeant une donnée recherchée, en suivant en sens inverse le chemin pris par la requête. Le point faible de cette technique est le nombre de messages importants émis pour une recherche, gaspillant inutilement la bande passante du système.

Par table de hachage distribuée. La localisation fonctionne indépendamment des réseaux physiques sous-jacents et constitue un réseau logique, nécessitant un mécanisme de nommage, tant pour les données que pour les pairs. Une fonction de hachage (par

exemple SHA-1) est appliquée sur l'adresse IP du nœud afin de générer les identifiants des pairs, notés *nodeID*. Les identifiants des données, notés *fileID*, sont générés à partir de la donnée elle-même en utilisant cette même fonction. Un pair est dit *responsable* d'une donnée si son *nodeID* est le plus proche du *fileID* de la donnée au sens d'une distance adéquate. Pour localiser une donnée, il suffit alors de router les requêtes vers le pair qui est responsable de la donnée recherchée en utilisant une *table de hachage distribuée* (DHT, pour *Distributed Hash Table*). Des exemples de couches de routage qui implémentent une telle approche sont Chord [163], Pastry [150] et Tapestry [191]. Ces mécanismes ont fait récemment l'objet d'améliorations basées sur la notion de graphe de De Bruijn. Ils ont abouti à l'élaboration de systèmes comme Koorde [103] et D2B [89]. Cette technique de routage permet de garantir de trouver une donnée dans le système de manière efficace. Notons que les implémentations actuelles de la spécification P2P JXTA [181] utilisent une variante de cette approche de routage par table de hachage distribuée, mais entre les super-pairs du système uniquement. Le chapitre 6 présente en détail JXTA.

La dernière catégorie de système a fait l'objet de nombreuses recherches par la communauté scientifique. L'abstraction des tables de hachage est utilisée pour bâtir des systèmes de stockage persistant de données. Des exemples de tels systèmes P2P sont CFS [64] et PAST [151].

4.2.3 Les limites

Les systèmes P2P, présentés à la section précédente, ont été étudiés essentiellement pour des applications de *partage de fichiers répliqués en lecture seule*. D'autres systèmes P2P visent le stockage des données de manière pérenne [160]. Toutefois, des systèmes comme OceanStore [111], Ivy [129] et dernièrement Pastis [46] ont pour but de proposer des mécanismes de partage de données modifiables visant la grande échelle[2]. Les deux derniers sont des prototypes de systèmes de fichiers P2P. Le premier est un système de gestion de données réparties et modifiables, mais qui offre également une interface de type système de fichiers.

OceanStore [111] utilise un modèle basé sur la résolution des conflits pour la gestion des mises à jour concurrentes. Les tentatives de modifications sont transmises à un ensemble de serveurs (en anglais *primary tier*) qui décident de l'ordre à appliquer pour ces modifications. Cet ensemble de serveurs est de cardinalité faible. Après avoir décidé de l'ordre des modifications, ils transmettent leur choix vers l'ensemble des serveurs possédant une copie. L'algorithme utilisé pour le choix des modifications est tolérant aux fautes byzantines, mais relativement coûteux. Par ailleurs, il nécessite que les serveurs qui participent à la décision soient bien connectés entre eux au niveau réseau et stables. Le nombre de modifications simultanées qui sont supportées est de ce fait faible et un nombre réduit de modifications simultanées suffit à diminuer les performances d'un tel système. Enfin, aucune garantie sur les mises à jour n'est assurée : les tentatives de modifications peuvent échouer. La comparaison avec NFS montre que le prototype d'OceanStore, appelé Pond, est globalement trois fois plus lent [147].

[2]Nous rappelons à notre lecteur que notre objectif n'est pas de faire un panorama détaillé, mais plutôt de présenter une vision globale.

Ivy [129] se base sur l'utilisation de fichiers appelés *log* pour le stockage des données. Chaque utilisateur du système possède un tel fichier dans lequel il écrit ses modifications sur l'ensemble des fichiers du système. Ces fichiers sont stockés dans la DHT qui repose sur le service de routage Chord. Pour avoir une vue cohérente du système de fichiers il doit parcourir tous les fichiers *log* des autres utilisateurs. Toutefois, l'utilisation de caches évite le parcours de la totalité de ces fichiers. Tout l'historique du système étant conservé, la détection et la résolution de conflit sont facilitées. Malgré tout, ce système ne peut supporter en pratique qu'un nombre très faible d'écrivains, ce qui constitue donc un frein à l'extensibilité du système et ne permet pas d'atteindre en pratique la grande échelle visée. Par ailleurs, le modèle de cohérence est similaire au modèle de cohérence *close-to-open* de NFS à la différence suivante : il n'y a pas de contrôle sur le moment où les modifications sont propagées. L'évaluation de Ivy montre des performances deux fois moindres par rapport à NFS. Enfin, il n'y a pas de gestion de la localisation des copies pour optimiser l'accès aux données.

Pastis [46] utilise des structures de données similaires à celles d'un système de fichiers classique. Ainsi, une structure appelée UCB (*User Certificate Block*) fait office de descripteur de fichiers (en anglais *i-node*). Les entrées d'un UCB pointent sur des CHB (*Content Hash Block*) qui représentent les blocs de données d'un fichier. Pastis utilise jusqu'à trois niveaux de tables d'indirection pour limiter la taille d'un UCB. Pastis se sert du système de stockage PAST pour stocker les UCB et les CHB. Un CHB est non modifiable et sa clé de stockage est un hachage de son contenu. En revanche, un UCB est modifiable et sa clé de stockage est une clé publique. La conception de Pastis s'inspire du système P2P CFS [64], mais diffère de celui-ci par l'utilisation de blocs modifiables (UCB) pour stocker les descripteurs de fichiers. Ainsi, à chaque modification d'une donnée, un nouveau CHB est créé et les UCB qui pointent dessus sont mis à jour. Cela permet d'éliminer l'effet de cascade jusqu'à la racine du système de fichiers lors de la mise à jour d'un descripteur de fichiers. Deux modèles de cohérence sont disponibles dans Pastis : le modèle *close-to-open* et *read-your-writes*. Ce dernier modèle offre des garanties de cohérence plus faibles. Il a nécessité de modifier la primitive de recherche des blocs de PAST, l'objectif était d'optimiser la récupération des copies en permettant de spécifier une version particulière d'une donnée. L'évaluation de Pastis montre qu'il est 1,4 à 1,8 fois plus lent que NFS (version 3). Le contrôle des accès concurrents à une même donnée se base sur un ordre total entre les processus, par utilisation de leur identifiant en cas de conflit.

En pratique, les résultats obtenus pour ces systèmes sont très loin d'atteindre la grande échelle visée. Les tests ont en effet été réalisés sur des systèmes constitués de seulement une dizaine de machines[3], ce qui ne peut pas être considéré comme de la grande échelle. Seul le système Pastis a été testé à plus grande échelle, jusqu'à 32 768 machines, mais par simulation. Notons que dans tous les cas ces systèmes supposent une topologie réseau tel que Internet, c'est-à-dire où les latences sont importantes et où leurs différences importent peu.

[3]Le maximum étant atteint par Ivy avec 32 machines répartis sur différents sites de PlanetLab [60].

	Transparence	Cohérence	Persistance	Tolérance à la volatilité	Performance
MVP	$\surd$	$\surd$	$\times$	$\times$	$\times$
P2P	$\sim$	$\sim$	$\surd$	$\surd$	$\sqcup$

TAB. 4.1 – Récapitulatif des propriétés des systèmes à MVP et des systèmes P2P. $\surd$: propriété satisfaite ; $\times$: propriété non satisfaite ; P : propriété partiellement satisfaite. $\sqcup$: propriété non évoquée.

4.3 Conclusion

Dans cette section, nous avons discuté des propriétés offertes par les systèmes à MVP et les systèmes P2P par rapport aux critères d'évaluations que nous avons définis à la section 3.3. Le tableau 4.1 récapitule nous conclusions. Nous pouvons constater que les systèmes à MVP permettent un partage cohérent de données de manière transparente. À l'opposé, les systèmes P2P permettent un stockage persistant des données malgré la volatilité des entités qui constituent ce type de systèmes. Nous avons indiqué que la propriété de performance n'est pas satisfaite pour les systèmes à MVP, compte tenu de leurs moins bonnes performances par rapport aux systèmes à base de passage de messages. Enfin, pour les systèmes P2P cette propriété est difficile à juger : les métriques sont trop disparates. Nous avons donc fait le choix d'indiquer que la propriété n'est pas évoquée, car difficilement évaluable. Toutefois, les performances d'un système P2P de partage de fichiers sont généralement moins bonnes qu'un système de fichiers classique tel que NFS.

Dans la partie suivante de ce manuscrit, nous présentons notre contribution pour la gestion de données sur les grilles de calcul qui est inspirée à la fois par les systèmes à MVP et P2P. Pour ce faire, nous avons défini un nouveau type de système que nous avons appelé *service de partage de données pour grilles*. Pour illustrer ce concept, nous proposons une architecture, que nous nommons JUXMEM, dont nous décrivons également la mise en œuvre.

Deuxième partie

Notre contribution : le service JUXMEM

Chapitre 5

JUXMEM : un service de partage de données pour grilles fondé sur une approche pair-à-pair

Sommaire

Nous introduisons dans cette deuxième partie du manuscrit notre contribution à la gestion de données pour grilles : le concept de *service de partage de données*. Nous en proposons une architecture appelée JUXMEM fondée sur une approche pair-à-pair (P2P). Dans le chapitre suivant, nous présentons la plate-forme JXTA qui nous a servi de support pour la mise en œuvre de JUXMEM. Enfin, le chapitre 7 décrit l'implémentation de notre architecture, en s'appuyant sur les fonctionnalités de JXTA.

La conception de la notion de service de partage de données pour grilles est issue de l'analyse conjointe des besoins des applications de calcul scientifique pour grilles (voir sections 2.4 et 3.1) et des systèmes existants (chapitre 3). Elle s'inspire des systèmes à mémoire

virtuellement partagée (MVP) et des systèmes P2P (voir chapitre 4). Les principaux aspects qui guident notre conception sont :

- une transparence de la localisation des données ;
- une gestion de la cohérence des données ;
- un stockage persistant ;
- une tolérance à la volatilité des équipements de l'infrastructure visée.

L'objectif de ce chapitre est d'introduire et de définir la notion de service de partage de données pour grilles, mais également d'en fournir une spécification en termes d'interface de programmation. En outre, nous présentons notre proposition d'architecture JUXMEM et nous justifions l'intérêt de sa mise en œuvre fondée sur une approche P2P.

5.1 Notion de service de partage de données

5.1.1 Sources d'inspiration

La présentation des systèmes de gestion de données pour grilles proposée dans le chapitre 3 a démontré l'inadéquation des systèmes actuels par rapport aux besoins des applications de calcul scientifique. Par exemple, la localisation des données partagées sur une grille de calcul reste généralement à la charge du code métier de l'application ou alors l'application doit supposer l'existence des données, sur les machines utilisées, préalablement à son exécution. Par ailleurs, la gestion de la cohérence des différentes copies des données partagées est également laissée à la charge du code métier de l'application par la quasi-totalité des systèmes actuels de gestion de données pour grilles. En outre, ces systèmes ne prennent généralement pas en compte dans leur conception la volatilité des entités sous-jacentes. Nous pensons que de telles limites ne permettent pas le passage à l'échelle de ces applications, en termes de nombre de machines utilisées pour leur exécution. De ce fait, les systèmes actuels de gestion de données pour grilles sont un frein au développement d'applications efficaces à grande échelle s'exécutant sur ce type d'infrastructure.

La présentation des deux approches pour la gestion de données en environnement distribué, proposée au chapitre 4 a toutefois permis de mettre deux types de systèmes aux résultats complémentaires : les systèmes à MVP et les systèmes P2P. En effet, les systèmes à MVP ont donné des résultats intéressants concernant les modèles et les protocoles de cohérence qui permettent de partager des données de manière transparente. À l'opposé, les systèmes P2P ont donné des résultats intéressants concernant le partage de données non modifiables. Si maintenant nous nous attachons à regarder les hypothèses de travail de ces systèmes, nous pouvons constater qu'elles sont opposées. Ces hypothèses sont déduites des caractéristiques des environnements cibles. Ainsi, les systèmes à MVP s'exécutent généralement sur une grappe de machines. Ce type d'environnement est statique, généralement homogène et avec peu de machines, entre une dizaine et une centaine. En revanche, les systèmes P2P supposent un environnement hétérogène, très volatile et à grande échelle, entre des milliers et des millions de machines. Un exemple typique est Internet.

Or comme nous l'avons décrit dans notre présentation des grilles de calcul (voir chapitre 2), notre environnement d'exécution se caractérise par :

- une échelle variant de 1 000 à 10 000 machines environ ;
- une volatilité de l'ordre de quelques heures ;

	Systèmes à MVP	Service de partage de données	Systèmes P2P
Échelle	Grappe de machines	Grille de calcul	Internet
Volatilité	Nulle	Moyenne (heures)	Forte (minutes)
Hétérogénéité	Nulle	Moyenne	Forte

TAB. 5.1 – Comparaison des hypothèses de travail entre les systèmes à MVP, notre concept de service de partage de données et les systèmes P2P.

– une hétérogénéité moyenne, c'est-à-dire limitée à quelques types de machines utilisées pour construire des grappes homogènes.

Nos hypothèses de travail sont donc intermédiaires par rapport à celles des systèmes à MVP et des systèmes P2P. Le tableau 5.1 résume cette comparaison.

À partir de l'analyse 1) de ce caractère intermédiaire de nos hypothèses de travail par rapport à celles émises dans les systèmes P2P et à MVP et 2) de la complémentarité des résultats des systèmes P2P et à MVP, notre contribution est de proposer un service de gestion de données pour le calcul scientifique sur grilles, *en s'inspirant* essentiellement *des points forts des approches des systèmes à MVP et P2P*. Pour les systèmes P2P, il s'agit de l'échelle atteinte et de la tolérance à la volatilité. En revanche, pour les systèmes à MVP, les points forts sont un espace mémoire global accessible par des entités réparties, et de façon transparente vis-à-vis de la localisation des données, ainsi que les modèles et protocoles de cohérence. Notre *approche hybride* a pour objectif de conserver les points forts des systèmes à MVP grâce à une conception fondée sur une approche P2P. Nous définissons une telle approche par le terme de *service de partage de données pour grilles* (en anglais *grid data-sharing service*).

5.1.2 Définitions

Avant d'énumérer la liste des propriétés que doit fournir notre service de partage de données pour grilles, nous définissons dans cette section ce que nous entendons par ce concept. Nous, commençons d'abord par préciser la notion de *service*. Les termes processus, applications et grilles de calcul correspondent à ceux introduits dans le chapitre 2.

Définition 5.1 : ***service*** — Interface de programmation mise à disposition des processus, appelés clients, par un ou plusieurs processus constituant des serveurs qui peuvent éventuellement collaborer pour fournir la fonctionnalité désirée.

Un service de partage de données se définit alors de la manière suivante.

Définition 5.2 : ***service de partage de données*** — Service dont la fonctionnalité est de fournir l'illusion d'un espace mémoire global pour le partage cohérent de blocs de données entre différents processus applicatifs.

Analysons cette définition. Un tel service vise à partager des blocs de données stockées en mémoire. Cette aspect est illustré sur la figure 5.1, où une partie de l'espace mémoire de deux processus de deux applications A et B est représentée. Nous utilisons dans la suite de ce manuscrit de manière indifférenciée les termes *blocs de données* et *données*. La taille visée

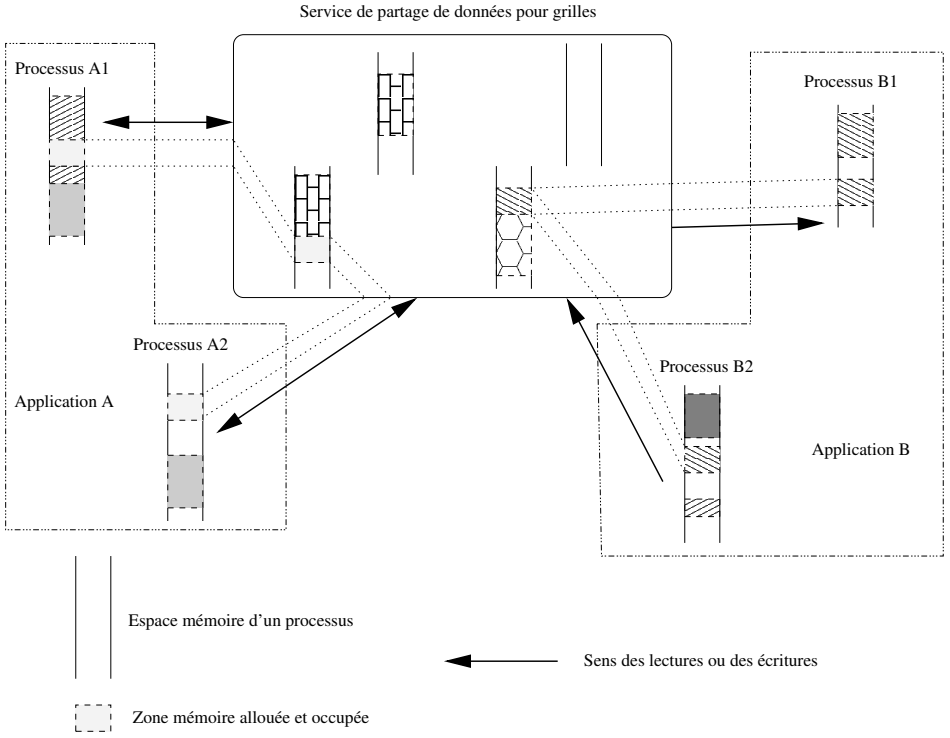

FIG. 5.1 – Illustration de la définition de notion de service de partage de données pour grilles.

pour ces blocs de données et de l'ordre de quelques dizaines à une centaine de mégaoctets. Les cases remplies (en grisé ou avec un motif) correspondent aux espaces mémoire où des données sont stockées. Ce schéma montre également l'illusion d'un *espace mémoire global*, que fournit un service de partage de données. Ainsi, les processus $A1$ et $A2$ de l'application A, accèdent au même espace mémoire du service de partage de données pour successivement y lire puis y écrire des données, comme l'indique le sens des flèches. Les processus $B1$ et $B2$ de l'application B, accèdent à un autre espace mémoire du service, pour respectivement y lire et écrire des données. Un partage d'espaces mémoire entre processus de différentes applications est également possible. Notons qu'un espace mémoire du service peut être projeté à différents endroits de l'espace d'adressage d'un processus applicatif.

Enfin, nous rappelons à notre lecteur que nous nous limitons à étudier cette définition et ses implications dans le contexte des grilles de calcul. En conséquence, les processus peuvent être distribués sur une ou plusieurs machines. Notre cas standard est d'ailleurs une distribution de l'ensemble des processus (applicatifs et du service) sur plusieurs sites d'une grille de calcul, avec un processus par machine.

5.1.3 Propriétés

Comme le précise la définition de la section précédente, le principal objectif de notre concept est de fournir un partage *transparent* et *cohérent* en mémoire de blocs de données par plusieurs processus applicatifs. Toutefois, d'autres propriétés sont également nécessaires afin de répondre aux besoins des applications mais aussi pour tenir de compte des

	Transparence	Cohérence	Persistance	Tolérance à la volatilité
Système d'origine	MVP	MVP	P2P	P2P

TAB. 5.2 – Systèmes d'origine pour les propriétés d'un service de partage de données.

contraintes imposées par l'environnement d'exécution. Celles-ci découlent des points forts des systèmes qui sont à la source de notre conception d'un service de partage de données. Cette section détaille chacune des propriétés de notre service de partage de données et le tableau 5.2 les résume, en indiquant leur principale source d'inspiration.

Transparence. Il s'agit de décharger les applications de la localisation et du transfert explicite des données entre les sites en ayant besoin. Le service localise les données et effectue automatiquement leurs transferts à la demande des processus applicatifs. Cette approche est à l'opposé de celle basée sur la *gestion explicite* des données, traditionnellement la plus suivie sur les grilles de calcul (voir chapitre 3). L'exemple le plus représentatif est Globus, présenté à la section 3.2.1, dans lequel l'utilisateur doit manuellement localiser ses données via des catalogues de copies et en gérer la cohérence. Cette propriété est directement inspirée par les systèmes à MVP. Le lecteur peut alors être amené à se demander quelles sont les différences avec un système à MVP ? La définition utilise le terme *espace mémoire global* et non *espace d'adressage global* comme dans les systèmes à MVP. En effet, compte tenu des possibilités qu'offre une grille de calcul, notamment son échelle, il apparaît illusoire de disposer d'un unique espace d'adressage, au sens système du terme, commun à tous les processus applicatifs.

Cohérence. Il s'agit de décharger les applications de la gestion de la cohérence des données partagées entre plusieurs processus applicatifs. Un utilisateur choisit un modèle et un protocole de cohérence pour une donnée et le service fournit les primitives nécessaires pour permettre des accès concurrents cohérents. Ainsi, pour reprendre l'exemple de la figure 5.1, le processus $B1$ souhaite lire la dernière valeur écrite par le processus $B2$. Cette propriété de cohérence de partage de blocs de données est nécessaire pour les applications de calcul scientifique sur grilles. La réalisation de cette propriété repose sur l'approche se basant sur les systèmes à MVP. En effet, les systèmes à MVP ont défini de nombreux modèles et protocoles de cohérence, tels que le modèle de cohérence à la libération, le modèle de cohérence à l'entrée et les protocoles associés reposant sur la notion de nœud-hôte (en anglais *home-based*) par exemple (voir section 4.1).

Persistance. Il s'agit de pouvoir réutiliser des données, afin d'effectuer un meilleur ordonnancement des calculs, compte tenu de leurs localisations. Le service permettant le partage de blocs de données en mémoire, nous entendons par persistance un temps de stockage supérieur à la durée de l'exécution d'une ou plusieurs applications qui ont besoin de ces blocs de données. L'objectif n'est donc pas de stocker les données de manière pérenne comme dans le système Us [160], par exemple. Cette propriété est issue des systèmes P2P, où par exemple un morceau de musique peut être disponible pendant de nombreux mois sur un réseau de partage de fichiers.

Tolérance à la volatilité. Il s'agit de tolérer les fautes qui peuvent avoir lieu dans l'environnement du service. Par exemple, une grappe de machines d'une grille de calcul peut

être arrêtée puis redémarrée suite à des problèmes techniques. L'architecture et l'implémentation d'un service de partage de données doivent être en mesure de tolérer cette volatilité des ressources. Ainsi, du point de vue des processus applicatifs, cette prise en charge d'éventuelles fautes doit être transparente et doit se caractériser tout au plus un par délai supplémentaire dans l'accès aux données. Par ailleurs, compte tenu de la propriété précédente de stockage persistant, des techniques adéquates de réplication des blocs de données sont nécessaires. Ainsi, les espaces mémoire au sein du service peuvent être répliqués. Cette propriété est également issue des systèmes P2P qui assurent la continuité de services qu'ils offrent malgré la volatilité des entités qui les composent.

Pour résumer, notre proposition ne vise pas à fournir un système à MVP pour la grille, ni un système P2P pour le partage de données modifiables. Notre objectif est bien de définir un nouveau type de système pour la gestion de données sur grilles via le concept de *service de partage de données*. Toutefois, c'est dans l'association des propriétés que se retrouve l'inspiration basée sur ces deux catégories de systèmes. L'objectif de notre système de gestion de données est de satisfaire les besoins des applications de calculs scientifiques s'exécutant sur cet environnement. L'ensemble de propriétés que doit fournir notre service va guider la conception de son architecture ainsi que son implémentation.

5.1.4 Spécifications

Avant de proposer une architecture pour la notion de service de partage de données pour grilles, nous nous définissons l'interface de programmation (en anglais *API*) qu'un tel service doit offrir aux processus applicatifs.

L'objectif de cette interface de programmation est double. Tout d'abord, nous souhaitons permettre une utilisation transparente des propriétés offertes par un tel service. Par exemple, la volatilité des ressources de stockage mémoire du service doit être gérée de manière transparente vis-à-vis des processus applicatifs. Ensuite, l'utilisation d'un tel service doit être simple et la plus naturelle possible.

5.1.4.1 Allocation mémoire

L'interface de programmation pour l'allocation et la libération d'espace mémoire est composée de 8 primitives. Afin de répondre aux objectifs précédemment mentionnés, l'interface de programmation proposée s'inspire des fonctions standard du langage C, familières aux développeurs d'applications. Dans la suite de cette section, nous détaillons ces ensembles de primitives en fournissant, à titre indicatif, leurs prototypes dans le langage C. Nous les présentons sous la forme de trois groupes. Le premier permet d'allouer et de libérer des espaces mémoire dans notre service de partage de données. Le deuxième ensemble permet : 1) de projeter une donnée partagée dans l'espace d'adressage de processus qui souhaitent y accéder et 2) de détruire localement cette projection. Enfin, le troisième et dernier groupe permet : 1) de projeter une donnée de l'espace d'adressage d'un processus dans des espaces mémoire au sein de notre service de partage de données et 2) de détruire localement cette projection. Ce dernier groupe de primitives n'est pas présenté dans cette section. Toutefois, l'annexe A.1 présente en détail l'ensemble des primitives de l'interface de programmation offerte par JUXMEM.

Dans notre proposition de service de partage de données, chacune des primitives permettant d'allouer un espace mémoire prend en entrée une liste variable d'arguments. Cette liste permet aux processus applicatifs de paramétrer la gestion de la donnée stockée dans l'ensemble des espaces mémoire alloués. Ces paramètres sont le protocole de cohérence, le nombre de sites à utiliser pour répliquer la donnée, le degré de réplication par site ainsi que le protocole de tolérance aux fautes associé au bloc de données. L'utilisation de ces paramètres en tant que contraintes (au risque d'échouer) ou suggestions est dépendant de l'implémentation, c'est-à-dire qu'elle ne fait pas partie des spécifications de notre service de partage de données. Enfin, l'ensemble des espaces mémoire alloués pour une donnée sont identifiés de manière globale et unique. Par abus de langage, nous appelons dans la suite de ce manuscrit cet objet un *identifiant de données*, que nous notons data_id.

void∗ juxmem_malloc(size_t size, char ∗∗data_id). Cette primitive permet d'allouer un ensemble d'espaces mémoire dans le service de partage de données.

void juxmem_free(char ∗data_id). Cette primitive permet de libérer l'ensemble des espaces mémoire alloués par le service pour le stockage de la donnée identifiée par data_id. La destruction de l'ensemble des copies n'est pas garantie si l'environnement d'exécution est soumis à des conditions de volatilité. Toutefois, l'accès à une copie d'une donnée libérée par cette primitive n'est plus possible. Un réinitialisation du processus qui stocke ce type de copies, dites *orphelines*, est alors nécessaire afin d'éviter la perte d'espace de stockage dans le service.

Le deuxième ensemble de l'API de notre proposition offre les primitives permettant d'accéder aux blocs de données partagées et déjà stockés dans notre service.

void∗ juxmem_mmap(void ∗ptr, size_t size, char ∗data_id, off_t off). Cette primitive permet de projeter à l'adresse ptr du processus applicatif les len octets de la donnée identifiée par data_id à partir de l'offset off.

void juxmem_unmap(void ∗ptr). Cette primitive détruit la projection effectuée à l'adresse ptr du processus applicatif d'une donnée partagée via le service. En outre, l'espace mémoire local est libéré. Toutefois, notons bien que l'ensemble des espaces mémoire utilisés pour stocker une donnée n'est pas réclamé par le service. Ainsi, la donnée est toujours accessible par d'autres processus applicatifs.

En trois phases, la figure 5.2 illustre l'utilisation de ces deux ensembles de primitives pour deux processus applicatifs A et B. Dans la première phase, le processus A alloue, via la primitive juxmem_malloc, un ensemble d'espaces mémoire dans le service de partage de données pour partager une donnée D (étape 1). Puis, le processus B récupère l'identifiant $id(D)$ de la donnée partagée D, par échange de messages par exemple. Une autre possibilité consiste à définir cet identifiant avant l'exécution de l'application. Ainsi, tous les processus de l'application peuvent le connaître. Le processus B va donc utiliser cet identifiant $id(D)$ pour projeter la donnée D dans son espace d'adressage via la primitive juxmem_mmap (étape 2). En fonction du protocole de cohérence spécifié pour ces espaces mémoire lors de l'appel à la primitive juxmem_malloc, les écritures seront propagées pour un partage cohérent. L'accès cohérent à la donnée partagée est garanti sous réserve que le programmeur suive le modèle d'accès mémoire tel qu'il est défini à la section 5.1.4.2. L'utilisation de la donnée partagée D constitue la seconde phase de l'exemple. Dans la troisième et dernière phase, le processus B détruit localement la projection de la donnée D dans son espace d'adressage, par l'intermédiaire de la primitive juxmem_unmap (étape 3). Enfin, le processus A utilise la

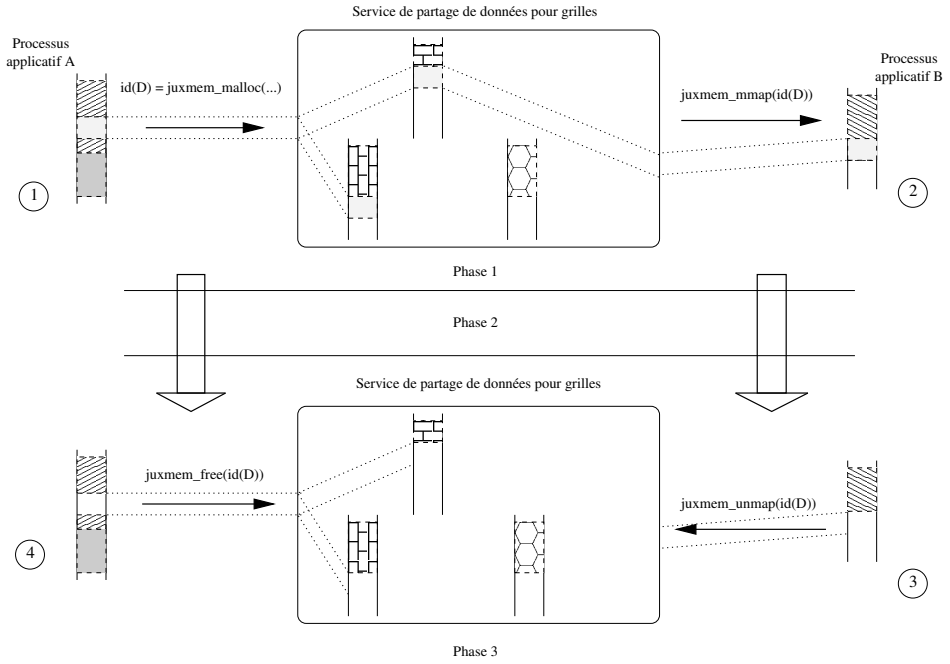

FIG. 5.2 – Enchaînement possible des primitives d'allocation et de destruction d'espaces mémoire pour le partage de données via JUXMEM.

primitive juxmem_free pour libérer l'ensemble des espaces mémoire qui stockent la donnée partagée D (étape 4).

5.1.4.2 Modèle d'accès mémoire

L'essence même d'un service de partage de données est de garantir la cohérence des différentes copies d'une donnée. Pour ce faire, une approche consiste à employer les modèles et les protocoles de cohérence développés dans le cadre des systèmes à MVP (voir section 4.1).

Notons d'abord que le choix d'un protocole de cohérence, et donc du modèle qu'il implémente, n'est pas imposé par notre service de partage de données. Comme nous l'avons vu dans la section précédente, un tel choix s'effectue à l'allocation de l'ensemble des espaces mémoire pour le stockage de la donnée partagée associée. Le choix revient aux développeurs d'applications. Un service de partage de données doit donc pouvoir offrir un assortissement varié des modèles de cohérence existants : forte ou relâchée par exemple.

Toutefois, compte tenu des besoins des applications de calculs scientifiques énumérés au chapitre 2, nous avons choisi le *protocole de cohérence à l'entrée* [41] (en anglais *entry consistency*, EC) comme protocole de référence. Il fait partie de la catégorie des modèles de cohérence relâchée qui ont fait l'objet d'une présentation détaillée à la section 4.1.2. L'utilisation d'objets de synchronisation pour la gestion des données partagées permet un gain de performance par élimination des mises à jour non nécessaires. Seuls les processus qui déclarent qu'ils vont accéder aux données seront mis à jour. De plus, ne seront mises à jour que les données qui seront effectivement modifiées. Enfin, ce protocole autorise plusieurs processus à lire en parallèle une même donnée (modèle *Multiple Readers Single Writer*, (MRSW) en

anglais). En effet, deux primitives permettent de distinguer les accès exclusifs pour écrire, des accès non exclusifs pour lire. Ces deux primitives ainsi que celle utilisée pour libérer un verrou sont décrites ci-dessous.

void juxmem_acquire(void *ptr). Cette primitive permet au processus applicatif d'acquérir l'accès exclusif à la donnée partagée, dont une copie est localement accessible via la pointeur ptr. Le processus peut ainsi modifier la donnée puisqu'il est assuré d'être le seul à avoir ce droit. Cette fonction est bloquante jusqu'à l'obtention du verrou dans ce mode : aucun autre processus n'a accès ni en lecture ni en écriture à cette donnée partagée.

void juxmem_acquire_read(void *ptr). Cette primitive permet au processus applicatif d'acquérir l'accès en lecture à la donnée partagée, dont une copie est localement accessible via la pointeur ptr. Le processus peut ensuite lire la donnée. Cette fonction est bloquante jusqu'à l'obtention du verrou dans ce mode : aucun autre processus ne doit plus avoir un accès en écriture à la donnée.

void juxmem_release(void *ptr). Cette primitive permet au processus applicatif de relâcher un verrou associé à la donnée partagée. Cette primitive peut être utilisée quel que soit le type du verrou, exclusif ou non exclusif. La donnée est identifiée dans l'espace d'adressage du processus local par le pointeur ptr. Le verrou ainsi libre sera attribué à l'un des éventuels processus applicatifs en attente. Le choix de favoriser les lecteurs ou les écrivains est laissé libre à l'implémentation.

Dans cette description, notons l'utilisation de pointeurs sur les espaces mémoire des processus applicatifs. En effet, dans notre modèle de programmation, un client peut projeter une même donnée partagée dans deux zones mémoire différentes de son espace d'adressage.

La figure 5.3 illustre deux processus qui utilisent ces primitives pour un accès cohérent à un tableau partagé identifié par data_id. Ainsi, le processus A crée et remplit initialement une donnée de a (code hexadécimal 0x61), sauf la dernière case pour en faciliter l'affichage. Puis, le processus boucle en écriture sur les 99 premières valeurs en incrémentant le code hexadécimal de chaque case. De son côté, le processus B projette la donnée partagée dans son espace d'adressage, avant de boucler en lecture sur les 99 cases de la donnée. Le code de synchronisation au démarrage des boucles n'est pas représenté sur la figure pour plus de simplicité. Cet exemple montre également clairement que chaque processus applicatif dispose dans son espace d'adressage d'une copie de la donnée partagée via notre service de partage de données. Il peut donc *lire et écrire directement*, par exemple à travers la primitive classique memset, ou via un mécanisme d'adressage par tableau comme dans l'exemple.

5.2 Conception fondée sur une approche pair-à-pair

Dans la section précédente, nous avons défini et spécifié la notion de service de partage de données. Comme nous l'avons constaté, les caractéristiques de l'environnement d'exécution visé sont intermédiaires par rapport à celles des systèmes à MVP et P2P. La conception de notre service de partage de données consiste à conserver les points forts des systèmes à MVP en s'appuyant sur des mécanismes P2P. En effet, ceux-ci ont prouvé leur extensibilité en termes de nombre de machines et leur tolérance à la volatilité, facteurs déterminants qui entrent en jeu dans la conception de notre service.

```
char *ptr = NULL;

ptr = juxmem_malloc(100, &data_id,
                    ...);
ptr = (char*) memset((void*) ptr,
                     0x61, 9);
ptr[99] = '\0';

for (i = 0; i < 99; i++) {
  juxmem_acquire(ptr);
  ptr[i]++;
  juxmem_release(ptr);
}
```

```
char *ptr = NULL;

ptr = juxmem_mmap(ptr, 0,
                  data_id, 0);

for (i = 0; i < 99; i++) {
  juxmem_acquire_read(ptr);
  printf("Read: %s\n", ptr);
  juxmem_release(ptr);
}
```

FIG. 5.3 – Illustration de l'utilisation des primitives pour l'accès cohérent à une même donnée par deux processus applicatifs.

Dans cette section, nous décrivons l'architecture que nous proposons pour notre service de partage de données pour grilles. Elle s'appelle JUXMEM, pour *Juxtaposed Memory*. Notons que nous restreignons au cas d'une grille formée d'une fédération de grappes distribuées, comme par exemple la grille expérimentale Grid'5000 présentée au chapitre 2.

5.2.1 Objectif et contraintes

L'objectif de JUXMEM est de minimiser les temps d'accès aux données partagées. Compte tenu des caractéristiques des grilles de calcul, deux contraintes se dressent face à cet objectif : 1) l'échelle grandissante de ces environnements et 2) la volatilité des machines formant une grille. En effet, la première contrainte nécessite par exemple la mise en place d'un algorithme de localisation d'une donnée partagée adapté à l'architecture des grilles de calcul. La deuxième contrainte, elle, peut ajouter des délais supplémentaires pour effectivement contacter l'entité responsable d'une donnée partagée. Compte tenu de ces contraintes, nous pensons que plusieurs aspects de JUXMEM peuvent être gérés en utilisant des algorithmes inspirés par les systèmes P2P. En effet, ceux-ci ont démontré pouvoir supporter ces contraintes et même les transformer en avantages pour améliorer l'efficacité des services qu'ils offrent.

Cet objectif de performance ne peut être atteint que par une prise en compte des caractéristiques physiques des grilles de calcul, notamment au niveau des performances des liens réseau. En effet, dans une grille de calcul, les machines d'une grappe d'un site sont interconnectées par des réseaux haute performance, appelé *System Area-Network* (SAN) en anglais. Ces réseaux ont une latence de l'ordre de quelques microsecondes et un débit pouvant aller à 20 Gb/s. Par exemple, un réseau Myrinet Myri-10G dispose d'une latence de 2 μs et d'un débit de 10 Gb/s dans ses dernières versions. En revanche, les différents sites sont connectés entre eux par des réseaux longue distance, en anglais *Wide Area Network* (WAN). Les caractéristiques de ces réseaux sont différentes de celles des SAN, en particulier sur le plan de la latence. Celle-ci peut varier d'une dizaine jusqu'à plusieurs dizaines de millisecondes, en fonction de la distance géographique séparant les sites d'une grille et de la charge réseau globale. Ainsi, il existe une différence d'ordre de grandeur pouvant aller de 2 000 à 25 000 pour

les latences si l'on compare un réseau SAN et un réseau WAN !

Cette topologie physique hiérarchique est prise en compte au niveau de JUXMEM par l'utilisation de la notion de *groupes*, issue des systèmes P2P. En effet, à chaque grappe de machines correspond un groupe appelé CLUSTER dont l'ensemble des processus s'exécutant sur cette grappe fait partie. La figure 5.4 représente trois groupes CLUSTER A, B et C au-dessus d'une grille constituée de trois grappes de machines A, B et C. La connaissance de la topologie réseau sous-jacente permet d'en tenir compte pour l'implémentation des protocoles de cohérence ou de gestion de verrous. L'objectif est alors de minimiser le nombre des coûteuses communications WAN et de privilégier les communications SAN à faible latence. Par ailleurs, la notion de groupe permet de repousser la contrainte sur l'échelle des grilles de calcul. En effet, la cardinalité maximale des groupes CLUSTER correspond à la taille d'un site d'une grille de calcul considérée. Cette cardinalité peut même être réduite par division d'un site en différentes sous-grappes virtuelles. Ainsi, le nombre d'entités entrant en jeu dans la gestion d'un tel type de groupe est limité, permettant la mise en place de mécanismes adaptés aux caractéristiques généralement homogènes sous-jacentes. De plus, la notion de groupe permet de limiter les effets de bord avec l'extérieur. En effet, la gestion interne d'un groupe n'est pas visible depuis l'extérieur. Par exemple, la volatilité des fournisseurs d'un groupe CLUSTER n'impacte pas celle de fournisseurs d'un autre groupe CLUSTER.

Toutefois, les différents groupes CLUSTER doivent communiquer entre eux. Ainsi, l'ensemble des processus présents dans JUXMEM font partie du groupe appelé JUXMEM, représenté à la figure 5.4. Ce groupe définit ce que nous appelons le réseau virtuel JUXMEM, ou plus simplement et par abus de langage, le réseau JUXMEM. Cet empilement hiérarchique des groupes présents dans JUXMEM est un élément supplémentaire pour repousser la contrainte sur l'échelle des grilles de calcul. En effet, l'hypothèse que nous pouvons faire sur la cardinalité du groupe JUXMEM et la répartition des processus permettent par exemple de choisir un mécanisme d'allocation mémoire adapté à la fois : 1) à la taille d'une grille et 2) à la présence de communications WAN entre les sites d'une grille. Cet algorithme est détaillé à la section 5.2.3.1. Notons, que pour poursuivre l'utilisation de termes issus des systèmes P2P, nous utiliserons de manière indifférenciée dans la suite de ce manuscrit le terme *pair* et le terme *processus*. Enfin, nous abordons la prise en compte de la contrainte de volatilité à la section 5.2.4.

5.2.2 Architecture générale

L'architecture générale de JUXMEM définit trois entités spécifiques pour les processus qui utilisent le service, appelés *clients*, le fournissent, appelés *fournisseurs*, ou participent à sa gestion, appelés *gestionnaires*.

Avant de détailler le rôle de chaque entité, attachons-nous à en décrire l'interconnexion. Les gestionnaires s'organisent en un graphe complet qui forme l'épine dorsale du réseau JUXMEM. En effet, chaque gestionnaire dispose d'une *vue locale* des autres gestionnaires présents dans le groupe JUXMEM[1]. Au bout d'un certain temps borné, les vues locales de chaque gestionnaire convergent vers le nombre exact de gestionnaires présents dans le réseau JUXMEM considéré. En revanche, les clients et les fournisseurs sont connectés à un gestionnaire

[1]Nous faisons l'hypothèse que la mise en œuvre de cette vue locale est fournie par l'exécutif P2P utilisé de manière sous-jacente.

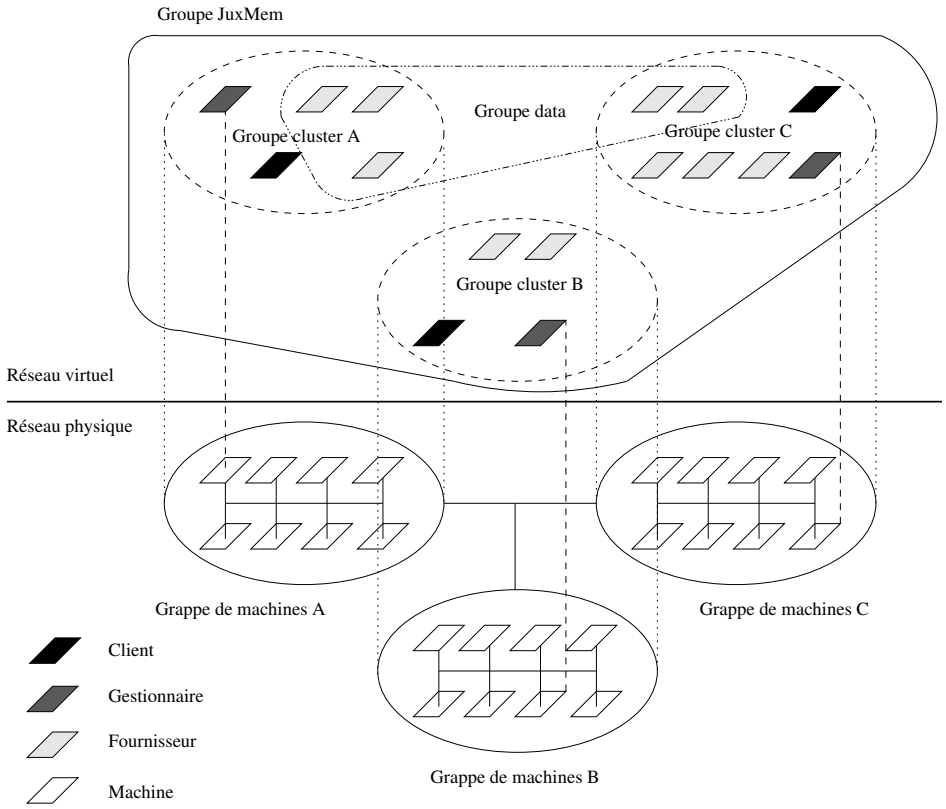

FIG. 5.4 – Hiérarchie des entités dans le réseau JUXMEM.

de leur choix. Ce mécanisme s'appelle un *abonnement* et permet aux clients d'utiliser notre service et aux fournisseurs de l'offrir. Notons qu'un abonnement n'est valide que pour une certaine période de temps et doit être régulièrement renouvelé par les fournisseurs et les clients. Enfin, dans un réseau JUXMEM, chaque entité fait à la fois partie du groupe JUXMEM global et d'un groupe CLUSTER.

Une description plus détaillée du rôle des trois entités de JUXMEM est donnée ci-dessous.

Client. Un client est un processus qui utilise le service de partage de données pour créer ou accéder à un ensemble de données. Un tel processus correspond généralement à un code applicatif et qui, pour les besoins de ses calculs, consomme, produit et/ou modifie des données. Toutefois, il peut également s'agir d'un intergiciel, comme défini au chapitre 2, sur lequel est conçu le code applicatif. Dans ce cas, cet intergiciel interagit avec le service de partage de données de manière transparente pour le code applicatif.

Fournisseur (en anglais *provider*). Un fournisseur est un processus qui offre à notre service de l'espace mémoire afin qu'il soit utilisé par des clients. Plus précisément, la disponibilité d'un espace mémoire est *publiée* dans le réseau JUXMEM vers le gestionnaire auquel est abonné le fournisseur. Les informations ainsi publiées sont le nombre de blocs mémoire offerts par le fournisseur, mais également l'identité logique du fournisseur. Un tel processus correspond à un démon qui s'exécute sur une machine physique. L'ensemble des fournisseurs du service définit la capacité du service en termes d'espace mémoire.

Gestionnaire (en anglais *manager*). Un gestionnaire est un processus qui gère un ensemble de fournisseurs et de clients, généralement proches physiquement. Chaque gestionnaire dispose d'un *cache local* pour pouvoir stocker les informations sur les espaces mémoire mis à disposition par ses fournisseurs. Un gestionnaire correspond à un démon qui s'exécute sur une machine d'une grappe et regroupe tout ou partie des fournisseurs de la grappe. Le rôle d'un tel processus est d'une part de fournir une vision agrégée de cet ensemble de fournisseurs et d'autre part de permettre aux clients d'utiliser les ressources mises à leur disposition sur un réseau JUXMEM. C'est donc un rôle structurel dans l'organisation du système.

La figure 5.4 illustre ces différentes entités présentes dans l'architecture de JUXMEM, ainsi que leur appartenance aux différents groupes introduits dans la section précédente. Notons qu'un processus peut être à la fois gestionnaire, client et fournisseur. C'est une caractéristique issue des systèmes P2P : la non-spécialisation des pairs d'un système pour une tâche et leur capacité à participer de manière simultanée dans les différents rôles possibles définis par le système. Toutefois, pour plus de clarté, chaque pair ne joue qu'un seul rôle sur la figure.

Par ailleurs, la figure 5.4 introduit un autre groupe dans JUXMEM : le groupe appelé DATA qui définit l'ensemble des fournisseurs qui hébergent une copie de la donnée partagée. Comme le montre la figure 5.4, ce type de groupe peut s'étendre sur un ou plusieurs groupes CLUSTER. En effet, une donnée peut être accédée depuis plusieurs sites. Le rôle d'un groupe DATA est de s'occuper de la gestion de la cohérence de la donnée associée. De plus, cette gestion doit prendre en compte l'hypothèse de volatilité des fournisseurs de notre service de partage de données. En conséquence, des mécanismes classiques de tolérance aux fautes sont mis en œuvre au sein de ces groupes DATA : détecteurs de défaillances, consensus, etc. L'originalité dans ces groupes réside dans la gestion conjointe de la tolérance aux fautes et de la cohérence des données. Elle s'appuie en effet sur des groupes auto-organisants qui ont pour objectif de s'adapter aux schémas d'accès ainsi qu'à la volatilité des fournisseurs. Ce travail constitue le cœur de la contribution de la thèse de Sébastien Monnet. De ce fait, ils ne seront pas plus détaillés dans ce manuscrit. Le lecteur intéressé peut se reporter à [127].

5.2.3 Gestion des ressources mémoire

Dans la suite de cette section, nous décrivons les mécanismes P2P utilisés dans JUXMEM pour l'allocation mémoire ainsi que l'accès à une donnée déjà existante.

5.2.3.1 Mécanisme d'allocation mémoire

Avant de décrire le mécanisme d'inspiration P2P pour l'allocation mémoire, voyons tout d'abord le format d'une requête émise par un client et sa sémantique. Un client peut exprimer dans sa requête le fait que :

> « *la donnée d'une taille x soit stockée dans n sites différents d'une grille, et qu'à l'intérieur de chaque site le nombre d'espaces mémoire utilisé soit m.* »

Trois paramètres entrent donc en jeu dans cette requête : 1) la taille x en octets de la donnée, 2) le nombre n de sites sur lesquels il faut répliquer la donnée et 3) le degré m de réplication intrasites. Ils permettent de contrôler le placement des espaces mémoire de manière

hiérarchique afin de correspondre à la topologie physique de l'infrastructure sous-jacente. Ces paramètres sont des entrées pour les primitives juxmem_malloc, juxmem_realloc et juxmem_attach définies à la section 5.1.4.1. L'objectif de ces paramètres est d'exprimer le fait qu'une donnée soit accessible depuis plusieurs codes de simulation d'une application dite de *couplage de code* (voir section 2.4). En effet, chaque code s'exécute sur une grappe et nécessite l'accès à un ensemble de données partagées entres les différents codes, justifiant l'existence du paramètre n. Le paramètre m est lui justifié par le besoin de tolérer $\lfloor \frac{m-1}{2} \rfloor$ fautes simultanées au sein d'une grappe[2].

L'allocation mémoire dans JUXMEM s'inspire des mécanismes de routage mis en place dans les systèmes P2P. Différents algorithmes ont été élaborés dans ce type de systèmes, tels que le routage avec répertoire centralisé [232], le routage avec répertoire distribué [138] et le routage par contenu [144, 191, 150, 164], *Distributed Hash Table* (DHT) en anglais (voir section 4.2). Cette dernière classe d'algorithme garantit de trouver une ressource dans un réseau avec une complexité faible en termes de saut logique, en $O(\log\ n)$. Toutefois, de tels algorithmes ne prennent généralement pas en compte la topologie physique sous-jacente. Ainsi, la complexité de ces techniques est traditionnellement évaluée en termes de nombre de sauts dans le réseau logique. Or, dans la topologie réseau hiérarchique d'une grille de calcul, le coût d'une communication à l'intérieur d'un site diffère du coût d'une communication intersite (d'un facteur entre 2 000 et 25 000 !). La prise en compte uniquement de la métrique de distance logique n'est pas suffisante dans ce contexte. La métrique de la distance physique entre les pairs est nécessaire et prédomine. De manière similaire à la section 5.2.1, l'objectif est alors de minimiser le nombre de communications intersites, compte tenu de leur coût élevé en termes de latence par rapport aux communications intrasites.

L'algorithme d'allocation d'espaces mémoire utilisé dans JUXMEM se fonde sur une variante du routage à répertoire distribué : l'approche à base de super-pairs qui agissent comme des index pour un ensemble fini de pairs. Dans notre cas, les super-pairs correspondent aux gestionnaires de JUXMEM. Le routage entre les gestionnaires se base sur la vue locale qu'ils ont des autres gestionnaires. Plus précisément, la décision d'orienter une requête d'allocation mémoire vers un gestionnaire plutôt qu'un autre se base sur la connaissance de l'état des autres gestionnaires. En effet, à cette vue locale est associé l'état des espaces mémoire offerts par chaque gestionnaire. Les informations sont la taille totale et le nombre de fournisseurs dans chaque groupe CLUSTER dont un gestionnaire a la charge. La mise à jour de ces informations se fait par inclusion de ces informations dans chaque message émis par le mécanisme de convergence des vues locales des gestionnaires (en anglais *piggybacking*). Cette solution permet d'obtenir un algorithme qui utilise deux ensembles de communications intersites en parallèle : une pour la propagation de la requête aux gestionnaires, puis une pour les réponses.

Exemple. La figure 5.5 illustre à travers un exemple les différentes étapes de cet algorithme. Elles sont par ailleurs décrites ci-dessous. Dans cet exemple, le client C émet une demande d'allocation mémoire avec comme paramètres : $x = 10$ (taille de la donnée), $n = 3$ (nombre de sites) et $m = 2$ (degré de réplication par site).

1. Le client du groupe CLUSTER $A1$ soumet sa requête d'allocation mémoire au gestionnaire $G1$ auquel il s'est abonné.

[2]Limite imposée par l'implémentation du protocole de consensus [56] utilisée dans JUXMEM.

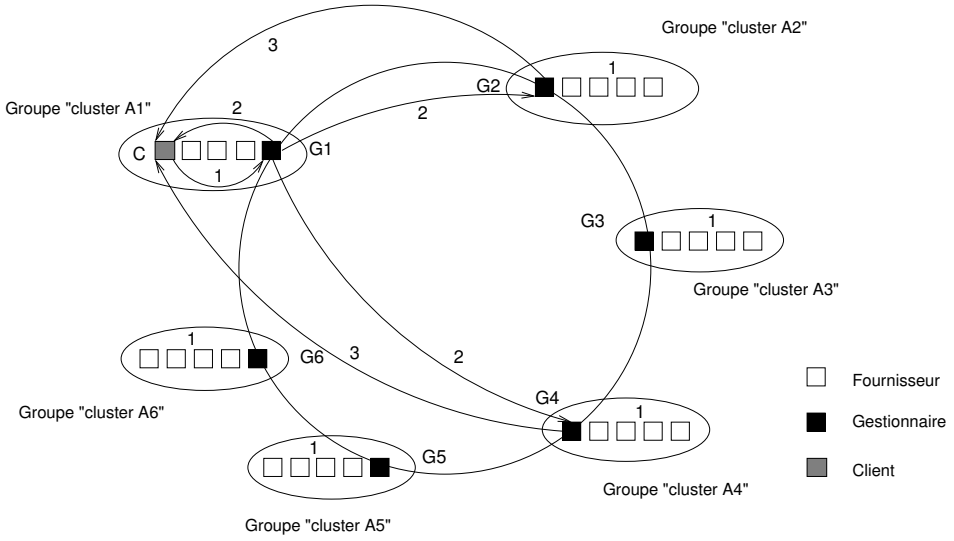

FIG. 5.5 – Étapes de l'algorithme d'allocation mémoire de JUXMEM.

2. Le gestionnaire $G1$ va alors transmettre la requête aux deux gestionnaires les plus pertinents $G2$ et $G4$ (car $n - 1 = 2$), toujours grâce à sa vue locale des autres gestionnaires. Par ailleurs, le gestionnaire $G1$ va retourner au client C une liste de deux fournisseurs (car $m = 2$) capables d'héberger une copie de la donnée, et mettre à jour son cache pour tenir compte de cette allocation.
3. De manière similaire, les gestionnaires $G2$ et $G4$ vont chacun retourner au client C une liste de deux fournisseurs capables d'héberger une copie de la donnée. En outre, ils vont mettre à jour leurs caches pour tenir compte de cette allocation.

Ainsi, le client dispose d'une liste de fournisseurs satisfaisant sa requête. Il va donc pouvoir demander à l'ensemble de ces fournisseurs de constituer un groupe DATA D. À chaque ensemble d'espaces mémoire alloués correspond un groupe DATA. Le client C va alors directement dialoguer avec ce groupe D pour l'accès à la donnée.

Par ailleurs, nous pouvons considérer dans ce mécanisme que les fournisseurs retournés au client sont réservés. Ainsi, si cette liste de fournisseurs n'est pas utilisée au bout d'un temps borné, les espaces mémoire réservés sont considérés comme libres et utilisables par une nouvelle exécution de l'algorithme. Enfin, notons que si la volatilité affecte des fournisseurs retournés au client, c'est à la charge du protocole de tolérance aux fautes de demander à trouver un nouveau fournisseur susceptible d'héberger une copie de la donnée.

5.2.3.2 Accès aux données partagées

Comme nous venons de le voir, l'allocation d'espaces mémoire dans JUXMEM va entraîner la création d'un groupe DATA pour assurer la gestion de la donnée partagée. L'identifiant de ce groupe correspond en réalité à *l'identifiant de la donnée*, tel que nous l'avons introduit à la section 5.1.4. Il est retourné au code applicatif en sortie de l'appel à la primitive juxmem_malloc. En outre, il est publié sur le réseau JUXMEM par l'ensemble des fournisseurs

du groupe DATA de la donnée. Cette publication s'effectue à la fois dans le groupe JUXMEM, mais également dans le ou les groupes CLUSTER ayant une intersection avec le groupe DATA de la donnée. L'objectif de cette publication est de rendre la donnée accessible à d'éventuels autres clients. Ainsi lors d'un appel à juxmem_mmap, notre service de partage de données va de manière automatique et transparente localiser la donnée désignée par son identifiant.

L'algorithme utilisé pour la localisation d'une donnée déjà existante dans JUXMEM s'effectue en trois temps.

1. Une recherche, dite *locale*, est lancée dans le groupe CLUSTER dont dépend le client qui souhaite accéder à la donnée. En effet, lorsqu'un client accède à une donnée il publie également l'identifiant de la donnée, en particulier sur le groupe CLUSTER auquel il est abonné. L'objectif est de trouver une copie proche pour optimiser le temps d'accès à une donnée. L'algorithme ne se bloque pas en attente d'une éventuelle réponse par un client ou un fournisseur.
2. Ensuite, une recherche, dite *globale*, est lancée dans le groupe JUXMEM. L'objectif est de trouver un ou plusieurs fournisseurs hébergeant un copie de la donnée, et cela quelle que soit la localisation de ces copies dans le réseau JUXMEM.
3. Enfin, l'algorithme se bloque jusqu'à recevoir une réponse par l'une ou l'autre des recherches. Lorsqu'une réponse est reçue, la primitive juxmem_mmap retourne au processus applicatif un pointeur sur la zone de la mémoire locale où se trouve une copie de la donnée. Si aucune réponse n'est reçue au bout d'un certain temps, par défaut de 10 secondes, la primitive retourne la valeur nulle.

Les primitives du modèle d'accès mémoire peuvent alors être utilisées pour accéder à la donnée (voir section 5.1.4.2).

5.2.4 Gestion de la volatilité

Dans cette section, nous décrivons les mécanismes P2P que nous avons mis en œuvre dans JUXMEM pour la gestion de la volatilité. Nous présentons tout d'abord les entités présentes dans JUXMEM, puis les communications nécessaires au bon fonctionnement de JUXMEM.

5.2.4.1 Au sein des entités

L'ensemble des entités de JUXMEM est soumis à la contrainte de volatilité imposée par les grilles de calcul. Toutefois, nous traitons de la volatilité de deux entités uniquement : les *fournisseurs* et les *gestionnaires*. En effet, nous faisons l'hypothèse que la volatilité des *clients*, c'est-à-dire des processus applicatifs, est à la charge de l'application ou de l'intergiciel sur lequel elle repose. Par exemple, des techniques adéquates de *point de reprise* et de *retour arrière* peuvent être mises en œuvre.

Volatilité des fournisseurs. La perte d'un fournisseur doit être prise en compte par le gestionnaire afin de mettre à jour sa connaissance sur l'espace mémoire offert par le groupe qu'il gère. Deux sous-cas peuvent se présenter, selon que la panne du fournisseur est imprévue ou non.

1. Dans le cas où elle est n'est pas imprévue, un message est envoyé sur le gestionnaire afin de l'informer du départ imminent du fournisseur. Le gestionnaire peut donc en tenir compte pour mettre à jour son cache local de la connaissance de l'espace mémoire total fourni. Par ailleurs, il peut également mettre à jour le nombre de fournisseurs qu'il gère.
2. En revanche, si la panne est imprévue, le mécanisme d'abonnement mis en place entre le fournisseur et son gestionnaire va permettre de détecter cet événement. En effet, en cas d'expiration de l'abonnement le fournisseur est considéré comme ne faisant plus partie du service. Le gestionnaire peut alors automatiquement en tenir compte pour mettre à jour son cache local de connaissance des espaces mémoire offerts.

Enfin, l'ajout d'un fournisseur est direct. En effet, la disponibilité de l'espace mémoire d'un fournisseur est publiée dès que le fournisseur est abonné à un gestionnaire.

Volatilité des gestionnaires. La perte d'un gestionnaire dans l'architecture présentée entraîne l'indisponibilité des fournisseurs mais également l'impossibilité pour les clients d'utiliser le service. La spécialisation d'un fournisseur en gestionnaire permet de prendre en compte la volatilité des gestionnaires. Ainsi, si les demandes de renouvellement d'abonnements échouent, les fournisseurs vont pouvoir détecter la perte du gestionnaire. Un mécanisme *d'auto-promotion* des fournisseurs en gestionnaires au bout d'un temps aléatoire mais borné est utilisé. Ce mécanisme est classique pour les systèmes P2P hiérarchiques à base de super-pairs. Dans notre cas, il va permettre de disposer à nouveau d'un gestionnaire auquel les fournisseurs vont pouvoir s'abonner. Notons que ce procédé peut aboutir à la présence de plusieurs gestionnaires pour un groupe CLUSTER. Toutefois grâce aux échanges périodiques de messages entre gestionnaires, le service va éliminer les éventuels doublons. En effet, lorsqu'un gestionnaire détecte d'autres gestionnaires pour un groupe CLUSTER, il va au bout d'un temps aléatoire mais borné redevenir un fournisseur.

5.2.4.2 Au sein des communications

Les communications entre les différentes entités du système doivent s'abstraire de la localisation physique des pairs dans le système. En effet, la volatilité des ressources mémoire doit être transparente pour les clients du service. L'utilisation d'un *identifiant logique*, pour désigner un canal de communication d'un client ou d'un fournisseur, va permettre de répondre à ce besoin de transparence pour dialoguer avec un fournisseur ou un ensemble de fournisseurs. Le système va dynamiquement résoudre à l'exécution cet identifiant logique en identifiant physique tel qu'un couple adresse IP et port.

Prenons ainsi l'exemple d'un client qui souhaite accéder à une donnée présente dans le service. La figure 5.6 illustre cet exemple, avec une donnée D répliquée sur 3 sites avec 2 copies par site ($n = 3$ et $m = 2$). Initialement, un tel client ne dispose que de l'identifiant global et unique de la donnée partagée. Le service doit « résoudre », en interne et de manière transparente, cet identifiant logique en un (ou plusieurs) identifiant(s) physique(s), afin que le client puisse utiliser la donnée partagée. Par ailleurs, ce mécanisme est également utile pour tolérer la volatilité des fournisseurs hébergeant une copie de cette donnée. Ainsi, si par suite d'une défaillance technique le groupe CLUSTER $A2$ disparaît, le système va être amené

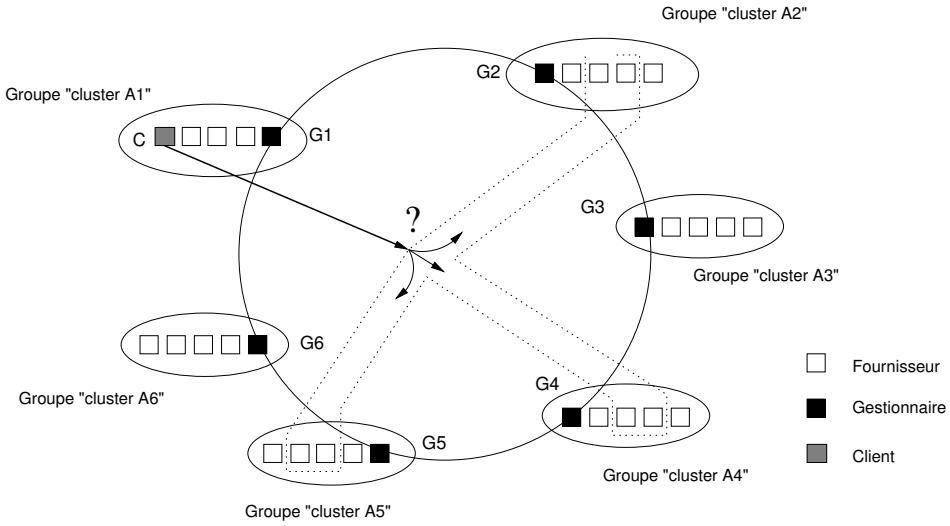

FIG. 5.6 – Illustration de la résolution dynamique des communications entre un client et un ensemble de fournisseurs hébergeant une copie de la donnée cherchée.

à répliquer la donnée sur un autre site afin de maintenir le degré de cohérence souhaité par le client. Afin de pouvoir dialoguer avec ces fournisseurs qui disposent d'une copie de la donnée, il est nécessaire de résoudre à nouveau l'identifiant logique de la donnée. Cette résolution doit être toujours transparente pour le client, qui doit juste noter une éventuelle latence supplémentaire lors de l'accès à la donnée partagée.

5.2.5 Discussion des choix de mise en œuvre

L'implémentation et la validation du concept de service de partage de données représentent une tâche longue et ardue. Pour faciliter ce travail, il semble naturel de s'appuyer sur un ensemble de briques de bases existantes. Dans les sections précédentes, nous avons montré la nécessité d'une approche P2P pour la conception et la mise en œuvre d'un tel service, et cela à travers la gestion des ressources mémoire et des communications qu'il est nécessaire de mettre en œuvre. Ainsi, nous avons été amenés à choisir un environnement P2P pour implémenter notre proposition d'architecture JUXMEM.

Le choix d'un environnement P2P particulier n'est pas une chose aisée, et la tentation d'en développer un, spécifique à nos besoins, peut être grande. Toutefois, nous ne pensons pas qu'une telle solution soit, à terme, viable. L'utilisation d'une plate-forme reconnue qui bénéficie d'un support via une communauté de développeurs conséquente est un gage de pérennité. En outre, cela nous permet de nous concentrer sur l'implémentation des propriétés de plus haut niveau souhaitées. Cependant, la prise en main et la compréhension des mécanismes internes d'une bibliothèque P2P sont des facteurs à prendre en compte pour ce choix. Par ailleurs, de telles bibliothèques souhaitent offrir un support pour la construction d'applications se basant sur les techniques P2P très variées. Il est donc nécessaire de pleinement les comprendre pour pouvoir les adapter à des besoins particuliers, comme par exemple leur utilisation sur des grilles de calcul. Plus concrètement, prenons quatre bibliothèques P2P existantes : JXTA-J2SE [221] (implémentation en Java de la spécification JXTA)

JXTA-C [179] (implémentation en C de la spécification JXTA), FreePastry [206] et Bamboo [148]. JXTA-J2SE (version 2.3.6) représente 187 713 lignes de code Java, JXTA-C (version 2.3) représente 90 497 lignes de code C, FreePastry 104 256 (version 1.4.3_02) lignes de code Java et Bamboo (version du 3 mars 2006) 42 517 lignes de code Java. Il faut donc trouver un juste milieu entre la complexité de la bibliothèque, le niveau d'abstraction offert et l'adéquation aux besoins requis.

Nous avons porté notre choix sur la spécification JXTA et ses implémentations dans les langages Java et C. Cette solution présente l'avantage de reposer sur une spécification claire et indépendante du langage de programmation. En outre, les spécifications JXTA fournissent des concepts de plus haut niveau, par rapport à des solutions comme FreePastry ou Bamboo. Par exemple, la notion de service de groupes existe dans les implémentations respectant la spécification JXTA. Un tel service est défini comme un service rendu par un ensemble de pairs qui peuvent éventuellement collaborer pour fournir le service en question. Un tel concept est en adéquation avec notre besoin pour l'implémentation du concept de service de partage de données. Par ailleurs, un réseau virtuel JXTA est configurable et les différents algorithmes de routage des systèmes P2P peuvent être mis en œuvre. Par exemple, l'intégration dans JXTA des mécanismes développés dans FreePastry ou tout autre système à base de tables de hachage distribuée (DHT) est possible et a déjà été réalisée [206]. Nous n'avons donc pas eu à redévelopper des primitives de bas niveau pour la gestion d'un réseau JuxMem. Enfin, JXTA spécifie la possibilité de disposer de communications dynamiques point-à-point ou par diffusion entre les pairs, fonctionnalité dont nous avons besoin. Le chapitre suivant décrit plus en détail la spécification JXTA et ses implémentations de références JXTA-C et JXTA-J2SE.

5.3 Discussion et conclusion

Ce chapitre a présenté notre approche pour la gestion des données dans les grilles, via la proposition du concept de service de partage de données.

L'objectif de ce service est de fournir l'illusion d'un espace mémoire global pour le partage cohérent de blocs de données entre différents processus s'exécutant sur une grille de calcul. Outre ce but de partage cohérent, nous avons défini les propriétés qu'un tel service doit offrir : transparence de la localisation des copies, persistance et tolérance aux fautes. De plus, l'obtention de ces propriétés est *transparente* pour les clients du service. Nous avons également précisé le modèle mémoire et l'interface de programmation d'un service de partage de données.

De nombreux raffinements ont été nécessaires afin de définir cette interface de programmation. Notons que cette interface est de bas niveau : elle ne considère des blocs de données que sous la forme d'une suite d'octets. Cela s'explique par le type d'application visé, des applications scientifiques où les données à partager sont généralement des tableaux ou des matrices d'octets. Toutefois dans un prolongement de ce travail, il serait intéressant de fournir une interface de plus haut niveau, au-dessus de l'interface actuelle, intégrant par exemple des données typées et structurées. Par ailleurs, il est compréhensible de s'interroger sur le choix d'une interface de programmation de type mémoire plutôt que de type fichier. Notre but étant de fournir un paradigme de programmation similaire aux systèmes à MVP, la réponse est relativement simple. En outre, il n'y a pas opposition entre ces deux interfaces de

programmation possibles, mais plutôt complémentarité. En effet, il est tout à fait possible de fournir l'interface de programmation de type fichier en s'appuyant sur une interface de type mémoire, via par exemple le projet FUSE [207]. Ainsi, les applications utilisant une interface de programmation par fichier pourraient, et cela de manière transparente, bénéficier des fonctionnalités de notre service de partage de données. Un travail dans ce sens a été initié mais n'est pas encore actuellement achevé et stable.

Dans un deuxième temps, nous avons introduit notre proposition d'architecture, appelée JUXMEM, s'inspirant des techniques P2P, pour la conception d'un tel service de partage de données. L'utilisation dès le début de telles techniques est nécessaire pour obtenir une architecture extensible et tolérante à la volatilité des ressources participant au service. Nous avons illustré un tel besoin sur la gestion des ressources mémoire et des communications entre les entités de l'architecture. Enfin, nous avons discuté et motivé notre choix d'une plate-forme P2P pour l'implémentation de JUXMEM : la spécification JXTA. Un tel choix nous est apparu clair au vu des fonctionnalités offertes par cet environnement et par la nécessité d'éviter un nouveau développement coûteux des mécanismes P2P nécessaires pour l'élaboration de JUXMEM. Les deux chapitres suivants présentent en détail la spécification JXTA et ses implémentations, puis son utilisation pour l'implémentation de JUXMEM.

Nous pouvons conclure que notre service de partage de données répond aux besoins des applications scientifiques qui s'exécutent sur les grilles de calcul : transparence d'utilisation, stockage persistant, partage cohérent et tolérance à la volatilité (voir section 3.1). Comme nous l'avons montré dans notre panorama des solutions utilisées pour la gestion des données sur les grilles de calcul (voir chapitre 3), aucun système actuel ne combine toutes ces propriétés de manière simultanée. Notre service fournit ces propriétés de manière transparente aux applications grâce une approche hybride s'inspirant des systèmes à MVP et des systèmes P2P (voir chapitre 4). L'objectif est de conserver les points forts des systèmes à MVP grâce à une conception fondée sur une approche P2P. Cette réponse aux besoins est dans la suite de ce manuscrit évaluée dans deux modèles de programmation utilisés sur les grilles de calcul (voir chapitres 8 et 9).

Chapitre 6

Présentation de l'environnement pair-à-pair JXTA

Sommaire

Dans ce chapitre, nous présentons l'environnement pair-à-pair (P2P) JXTA : sa spécification et ses implémentations de référence JXTA-J2SE [221] et JXTA-C [179]. Cette présentation est justifiée en raison de l'emploi de cette bibliothèque P2P pour l'implémentation de l'architecture JUXMEM décrite au chapitre précédent. Dans un premier temps, nous décrivons de manière générale les objectifs de JXTA et nous définissons ses principaux concepts. Puis, après un bref aperçu des protocoles et des services JXTA, nous présentons de manière plus détaillée trois services au cœur de l'architecture de l'environnement. Tout d'abord le *service de rendez-vous*, puis le *service de découverte* et enfin le *service de canal virtuel*. La présentation en détail de ces trois services est motivée par leur utilisation pour la mise en œuvre de JUXMEM, qui fait l'objet du chapitre suivant.

6.1 Présentation générale

L'objectif de cette section est de présenter de manière générale l'environnement P2P JXTA [181]. Le nom JXTA vient du verbe anglais *to juxtapose* et indique que le modèle P2P n'est pas à considérer en opposition par rapport au modèle client/serveur. Au contraire, il est complémentaire de ce dernier.

Tout d'abord, précisons les termes que nous utilisons. JXTA est une *spécification des formats d'un ensemble de protocoles*, appelés noyau (en anglais *core*) et de concepts pour la construction d'applications P2P [222]. Elle contient en outre :

- la définition d'un ensemble de protocoles et de services standards optionnels, qu'il est souhaitable que tout pair JXTA implémente ;
- les étapes conseillées d'une connexion entre pairs pour trois protocoles de transport (TCP/IP, HTTP et TLS) ;
- ainsi que la représentation réseau standard du format des messages JXTA (en-têtes et structure).

La dernière modification apportée à cette spécification date du 20 septembre 2005 et son numéro de version est 2.3.5. Il existe plusieurs implémentations plus ou moins avancées de ces spécifications. JXTA-C et JXTA-J2SE, écrites respectivement dans le langage C et Java, sont les plus abouties. L'évolution majeure de la version 2.0 de JXTA-J2SE a été publiée le 3 mars 2003 et sa version actuelle est la 2.4. La version 2.0 de JXTA-C a été publiée quant à elle le 16 juillet 2004 et sa version actuelle est la 2.5. Il ne faut pas confondre JXTA avec son implémentation JXTA-J2SE, longtemps la seule à respecter la dernière version des spécifications. Enfin, lorsque nous utilisons les termes *environnement* JXTA ou *plate-forme* JXTA, nous faisons référence à la fois à la spécification et à ses implémentations.

L'environnement JXTA est le fruit d'un projet de recherche de Sun Microsystems dirigé par Bill Joy, ancien directeur de la recherche chez Sun Microsystems et cofondateur de l'entreprise, et de Mike Clary. Le projet a été rendu public le 25 avril 2001 et est depuis dirigé par Bernard Traversat, directeur du groupe « développements avancés » chez Sun Microsystems. C'est un projet libre, sous une licence de type Apache légèrement modifiée, et qui dispose d'une forte communauté d'utilisateurs et de contributeurs actifs.

6.1.1 Objectifs

L'objectif de la spécification JXTA, ou plus simplement JXTA, est de fournir les mécanismes de base pour la construction de systèmes P2P. De tels mécanismes sont par exemple, la communication entre les pairs, la découverte de ressources, etc. JXTA se veut donc être une plate-forme générique pour l'élaboration de systèmes P2P, indépendante des architectures matérielles et des langages.

Ainsi, JXTA ne définit ni l'algorithmique à utiliser pour l'implémentation des protocoles, ni même les interfaces de programmation associées, mais uniquement le format des messages émis par chaque protocole spécifié. Les implémentations de la spécification JXTA sont de ce fait libres d'utiliser une technique particulière, par exemple, pour la découverte de ressources sur un réseau virtuel ou la propagation des messages. C'est là toute la force de JXTA : une plate-forme générique qui peut servir de cadre à l'évaluation de nouveaux algorithmes P2P.

Par ailleurs, JXTA offre la possibilité aux applications P2P d'être interopérables et permet une réduction du coût de développement de tels systèmes. En effet, les principaux systèmes actuels P2P, notamment les systèmes de partage de fichiers, reposent sur des protocoles spécifiques et généralement propriétaires. Or, le temps de développement de tels protocoles est relativement long et donc coûteux. JXTA permet de bâtir une application P2P sur des briques de base déjà existantes. En outre, les différents systèmes peuvent plus simplement collaborer puisqu'ils reposent sur la même plate-forme.

Enfin, l'indépendance vis-à-vis des architectures matérielles et des langages permet de mettre en œuvre des implémentations de JXTA sur n'importe quel équipement électronique : depuis l'assistant personnel au serveur, en passant par la machine de bureau. En outre, toutes ces implémentations sont interopérables : une implémentation C est à même de dialoguer avec une implémentation Java ou Python d'un système P2P basé sur JXTA.

6.1.2 Définitions

Avant de présenter les protocoles définis par JXTA, nous introduisons tout d'abord les principaux concepts auxquels ils font référence.

Définition 6.1 : ***pair*** — Entité de base d'un réseau JXTA, identifiée de manière unique par une adresse logique, appelée *pair ID*, indépendante de sa localisation dans le réseau physique. Un pair correspond à un processus qui implémente les protocoles noyau de JXTA.

Nous verrons dans la suite de ce chapitre à quoi correspondent ces protocoles noyau. JXTA ne spécifie pas différents types de pairs, mais en permet l'existence. Ainsi en pratique et dans les implémentations JXTA-C et JXTA-J2SE, trois types de pairs existent : les pairs de type *standard*, les pairs de type *rendez-vous* et les pairs de type *relais*. Pas abus de langage et par souci de concision nous les appellerons respectivement pairs standards, pairs rendez-vous et pairs relais. Les caractéristiques de ces pairs au sein des implémentations JXTA-C et JXTA-J2SE sont décrites ci-dessous.

Pairs standards (en anglais *full-featured edge peers*). Pairs qui peuvent émettre et recevoir des requêtes. Ils disposent également d'un cache local, généralement sous la forme d'une base de données. Ce cache permet de stocker des informations sur les ressources qu'ils ont découvertes sur le réseau P2P. Une version « allégée » de ce type de pair existe, appelée pair standard minimal (en anglais *minimal edge peer*). Elle ne dispose pas de cache local et est particulièrement adaptée pour s'exécuter sur des assistants électroniques par exemple.

Pairs rendez-vous (en anglais *rendezvous peers*). Ils disposent des mêmes caractéristiques que les pairs standards. Ils sont par ailleurs responsables de l'indexation des ressources présentes sur le système P2P pour permettre aux pairs de les trouver efficacement. Enfin, ils ont la charge de propager les messages dans le système. Les pairs rendez-vous sont donc assimilables à des super-pairs.

Pairs relais (en anglais *relay peers*). Ils disposent des mêmes caractéristiques que les pairs standards. Il sont toutefois capables de stocker temporairement des messages pour d'autres pairs. Ainsi, ils permettent de passer outre les pare-feux et autres systèmes de translation d'adresse (NAT). En effet, un pair s'exécutant derrière un pare-feu n'est pas accessible depuis l'extérieur, sauf s'il initie les connexions, en particulier vers un pair relais. Ce dernier va alors lui transmettre ses messages en attente.

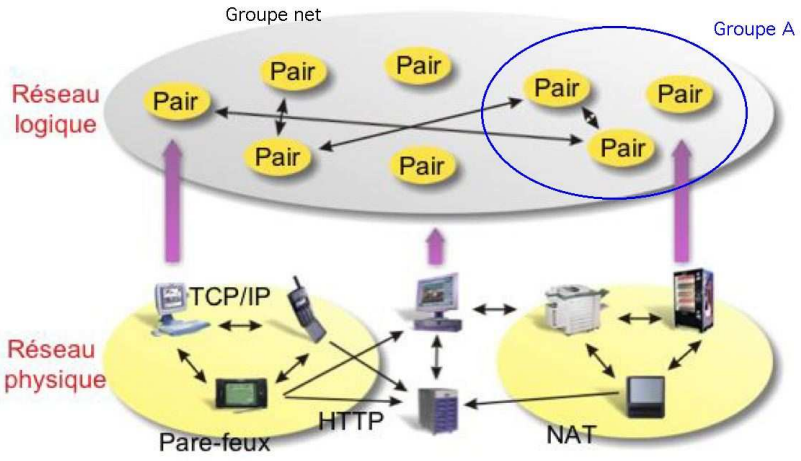

FIG. 6.1 – Réseau virtuel JXTA au-dessus d'un réseau physique.

Notons qu'un même pair peut être à la fois pair rendez-vous et pair relais.

Définition 6.2 : ***groupe de pairs*** — Regroupement de pairs selon des intérêts communs qui dispose d'un identifiant unique appelé *groupe ID*.

Un groupe définit un domaine, éventuellement sécurisé, au sein duquel le regroupement de pairs peut s'échanger des messages. De plus, dans un domaine d'autres groupes peuvent se former. Une hiérarchie de groupes peut être ainsi bâtie pour raffiner un regroupement de pairs. Un pair peut être membre de plusieurs groupes à la fois. Par ailleurs, il peut jouer des rôles différents (standard, rendez-vous, etc.) dans chacun des groupes dont il est membre. Un type de pair se définit donc par rapport à un groupe. Enfin, tous les pairs JXTA sont créés dans un groupe prédéfini, le groupe *net*. Il permet à tous les pairs JXTA d'être en mesure de s'échanger des messages. L'identifiant de ce groupe est modifiable afin de créer un groupe *net* privé et propre à un ensemble de pairs utilisant cet identifiant. La figure 6.1 représente un réseau virtuel JXTA au-dessus d'un réseau physique. Deux groupes sont représentés sur la figure. Tous les pairs font partie d'un groupe *net*. Trois pairs sont en plus membres du groupe A, représenté en bleu sur la figure.

Définition 6.3 : ***annonce (en anglais advertisement)*** — Document XML qui décrit, sous la forme de métadonnées, une ressource disponible sur un réseau JXTA.

Il existe plusieurs types d'annonces prédéfinies par JXTA : les annonces de pairs, de groupes, etc. Les protocoles JXTA utilisent fréquemment de tels documents dans les messages que s'échangent les pairs. Chaque annonce est publiée via le protocole de découverte (voir section 6.2.2) avec une durée de vie précise (qui peut être étendue). Cette publication se fait toujours dans le contexte d'un groupe. Toutefois, une même annonce peut être publiée dans un ou plusieurs groupes.

Définition 6.4 : ***canal virtuel (en anglais pipe)*** — Moyen de communication entre deux ou plusieurs pairs, identifié de manière unique par un *pipe ID* et indépendant de la localisation des pairs d'un groupe JXTA l'utilisant.

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
<Id>urn:jxta:uuid-5961629B8021441095C3AB1101C76DCC ... 04</Id>
<Type>JxtaUnicast</Type>
<Name>PipeTest</Name>
</jxta:PipeAdvertisement>
```

Figure 6.2 – Annonce d'un canal virtuel de type unidirectionnel ayant pour nom PipeTest.

Un canal virtuel permet à un ensemble de pairs désirant communiquer de s'abstraire de leur localisation physique mais également de la topologie réseau. Ainsi, ils disposent d'une communication logique directe, même si de manière sous-jacente de multiples sauts peuvent être faits, des pare-feux traversés et différents protocoles de transports réseau utilisés. Le chapitre 10 présente plus en détail le service de canaux virtuels. La figure 6.2 représente l'annonce d'un canal virtuel de type unidirectionnel ayant comme nom PipeTest. Un des protocoles définis par JXTA, le protocole de liaison de canal virtuel, permet de « résoudre » les deux extrémités d'un canal virtuel pour les associer physiquement à des pairs. Sur le pair émetteur, un canal virtuel en sortie (en anglais *output pipe*) est créé afin d'envoyer les messages. Du côté du pair récepteur, un canal virtuel en entrée (en anglais *input pipe*) est créé pour pouvoir recevoir les messages émis.

Définition 6.5 : ***service réseau*** — Interface de programmation offrant une ou des fonctionnalités particulières dans un groupe d'un réseau JXTA.

JXTA ne spécifie pas de manière plus précise la notion de service réseau, notamment la manière de les invoquer. L'objectif est d'être interopérable et compatible avec les standards actuels et futurs des services Web, tel que WSDL par exemple. Un service réseau dit JXTA est un service réseau décrit par une annonce de type *spécification de module*. Une telle annonce peut notamment contenir un identifiant de canal virtuel pour permettre d'utiliser le service.

Toutefois, JXTA distingue deux types de service réseau : le service de pair et le service de groupe.

Service de pair. Un service réseau de pair est accessible uniquement sur le pair qui a publié ce service et cela dans les groupes à partir desquels la publication s'est faite. Si le pair n'est plus disponible, le service n'est plus utilisable.

Service de groupe. Un service de groupe est quant à lui composé d'une collection d'instances de ce service s'exécutant sur plusieurs pairs, qui peuvent éventuellement coopérer pour fournir le service. Les services de groupes ne doivent pas en principe être sensibles à la défaillance d'un des membres du groupe.

Notons que la notion de groupe est centrale dans JXTA, elle borne l'utilisation des services et crée une frontière avec l'extérieur. À l'intérieur de cet espace est défini l'ensemble des services que chaque pair membre du groupe doit fournir. Parmi ces services de groupe, JXTA en définit certains qui ont pour objectifs d'implémenter le format réseau des messages émis ou reçus par les protocoles spécifiés par JXTA.

La notion de service est également utilisée pour implémenter les protocoles définis par les applications basées sur JXTA. Par exemple, un protocole d'accès à une ressource peut

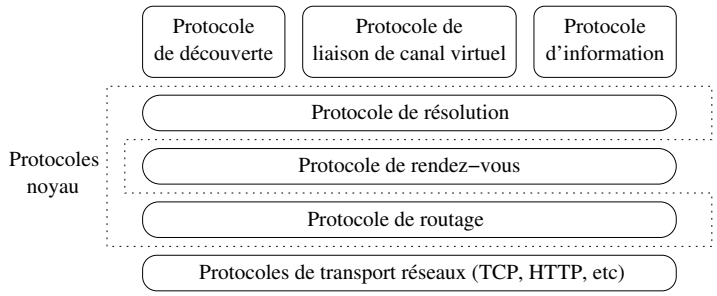

FIG. 6.3 – Pile des protocoles définis par la spécification JXTA.

être implémenté sous la forme d'un service de groupe. Il est important de comprendre que la portée d'utilisation d'un service implémentant un protocole (JXTA ou applicatif) se limite à l'ensemble des pairs membres du groupe dans lequel le service s'exécute.

6.1.3 Un aperçu des protocoles et des services JXTA

La spécification JXTA définit le format des six protocoles qui travaillent ensemble pour la découverte, l'organisation, la surveillance et les communications entre les pairs. Ces protocoles ainsi que leurs rôles sont décrits ci-dessous. La figure 6.3 présente la hiérarchie des protocoles JXTA, leurs dépendances ainsi que leur appartenance aux protocoles noyau de JXTA ou non. Les protocoles noyau sont ceux qu'il est impératif qu'un pair implémente pour qu'il respecte la spécification JXTA. Chaque protocole décrit dans cette section est implémenté sous la forme d'un service réseau de groupe.

Protocole de routage (en anglais *Endpoint Routing Protocol*). Ce protocole permet à un pair de découvrir la route à suivre pour envoyer un message à un autre pair. Ce mécanisme est utilisé lorsqu'il n'existe pas de route directe entre deux pairs souhaitant communiquer. Il va alors permettre de trouver le(s) pair(s) par lequel le message doit transiter. Ce protocole est l'un des deux protocoles noyau.

Protocole de rendez-vous (en anglais *Rendez-Vous Protocol*). Ce protocole permet aux pairs standards de s'abonner à un service de réception des messages qui sont propagés dans un groupe JXTA. Il permet également de propager des messages aux pairs membres d'un groupe. Enfin, il permet d'organiser la structure du réseau logique des pairs rendez-vous et cela toujours dans le contexte d'un groupe. Ce protocole repose sur le protocole de routage pour l'envoi et la réception de messages.

Une description plus détaillée du service implémentant ce protocole (service de rendez-vous, appelé rendezvous dans le code des implémentations) et de son fonctionnement interne est proposée en section 6.2.1

Protocole de résolution (en anglais *Peer Resolver Protocol*). Ce protocole permet d'envoyer des requêtes à un ou plusieurs pairs d'un groupe JXTA. Zéro, une ou plusieurs réponses par requête peuvent également être retournées via ce protocole. Ce protocole a

pour objectif de permettre l'implémentation de services de plus haut niveau, applicatifs par exemple. C'est à ce niveau de la spécification des protocoles qu'est introduite la possibilité pour un pair de participer à un mécanisme d'index distribué des ressources partagées, en anglais *Shared Resource Distributed Index* (SRDI). Un tel index est réparti au sein d'un groupe sur le réseau de pairs rendez-vous et permet aux requêtes d'être mieux guidées dans leur diffusion. Le protocole de résolution repose sur le protocole de rendez-vous pour l'envoi et la réception de messages.

Le service de résolution (appelé resolver dans le code des implémentations) implémente ce protocole. Ce protocole est le deuxième protocole noyau.

Protocole de découverte (en anglais *Peer Discovery Protocol*). Le protocole de découverte repose sur le protocole de résolution pour l'envoi de ses requêtes et des réponses associées. Ce protocole permet aux pairs de mettre à disposition leurs ressources sur un réseau JXTA, mais également de les découvrir. Comme nous l'avons vu précédemment, chaque ressource est décrite par une annonce. Chaque type d'annonce possède des attributs qui sont utilisés lors de la publication ou de la recherche de ces documents associés aux ressources.

Le fonctionnement interne de ce service (service de découverte, appelé discovery dans le code des implémentations), notamment l'utilisation du mécanisme d'index distribué pour le placement des index des annonces sur le réseau de pairs rendez-vous d'un groupe, est détaillé à la section 6.2.2.

Protocole de liaisons de canal virtuel (en anglais *Pipe Binding Protocol*). Ce protocole permet à un pair d'établir un canal de communication virtuel avec un ou plusieurs pairs. Plus précisément, il est utilisé pour mettre en relation les extrémités d'un canal de communication par leur(s) résolution(s) en adresses réseau. C'est le mécanisme au cœur du fonctionnement des communications entre pairs dans la spécification JXTA. Ce protocole utilise le protocole de résolution pour déterminer le(s) pair(s) en écoute sur l'extrémité d'un canal.

Le fonctionnement de ce service (service de canaux virtuels, appelé pipe dans le code des implémentations) est détaillé à la section 6.2.3.

Protocole d'information (en anglais *Peer Information Protocol*). Ce protocole permet à un pair d'obtenir des informations à propos des autres pairs d'un réseau JXTA. De telles informations peuvent être la charge réseau, la durée de vie du pair, ses capacités, etc. Il repose également sur le protocole de résolution.

En pratique, ce protocole n'est que partiellement supporté par JXTA-J2SE et n'est donc pas plus détaillé.

6.2 Présentation des services de base de JXTA

Dans cette section, nous présentons plus en détail l'implémentation, en tant que service, de trois protocoles au cœur de JXTA : le service de rendez-vous, le service de découverte, le service de canal virtuel. Nous tenons à faire remarquer à notre lecteur que les choix d'implémentations présentés dans cette section ne font pas partie de la spécification JXTA. Par

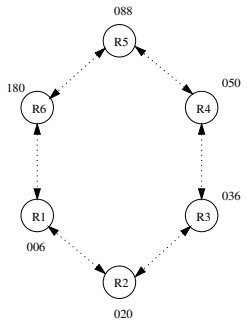

FIG. 6.4 – Représentation de l'anneau formé par les pairs rendez-vous.

ailleurs, cette présentation est issue de lectures du code de JXTA-J2SE et JXTA-C qui représentent respectivement 187 713 et 90 497 lignes de code (!). En effet, l'algorithmique utilisée pour l'implémentation des protocoles JXTA n'est généralement pas documentée.

6.2.1 Service de rendez-vous

Le service de rendez-vous tel qu'il est défini par la spécification JXTA inclut trois sous-protocoles : 1) le *protocole de gestion de vue* (en anglais *peerview protocol*) utilisé par les pairs rendez-vous pour organiser leurs interconnexions, 2) le *protocole d'abonnement* aux pairs rendez-vous (en anglais *rendez-vous lease protocol*) utilisé par les pairs standards pour s'abonner à la réception de messages propagés par le service et 3) le *protocole de propagation* par les pairs rendez-vous (en anglais *rendez-vous propagation protocol*) qui permet de gérer la diffusion des messages au sein d'un groupe.

Les deux premiers sous-protocoles sont optionnels et ne disposent pas d'interfaces de programmation externe.

Protocole de gestion de vue. Ce protocole permet aux pairs rendez-vous de travailler ensemble à former une vue globale des pairs rendez-vous (en anglais *peerview*). Cette vue globale est toujours relative à un groupe JXTA. Toutefois, chaque pair rendez-vous du groupe maintient une version locale de cette vue globale (appelée *vue locale*). Elle est utilisée pour router les messages. Le but du protocole de gestion de vue est de construire et de maintenir cohérentes ces vues locales sur l'ensemble des pairs rendez-vous d'un groupe. L'algorithmique interne de ce protocole est détaillée à la section 11.3.1. Outre cette vue locale, chaque pair rendez-vous maintient un pointeur vers : 1) le pair rendez-vous dont l'identifiant est le plus grand ID strictement inférieur à l'identifiant du pair rendez-vous local et 2) le pair rendez-vous dont l'identifiant est le plus petit ID strictement supérieur à l'identifiant du pair rendez-vous local. En conséquence, les pairs rendez-vous forment un anneau, c'est-à-dire deux graphes circuits : l'un dans le sens lexicographiquement croissant des identifiants des pairs, et l'autre dans le sens inverse (voir figure 6.4).

Protocole d'abonnement. Ce protocole permet à des pairs standards de se connecter à des pairs rendez-vous afin d'utiliser le réseau virtuel JXTA d'un groupe donné. Les pairs rendez-

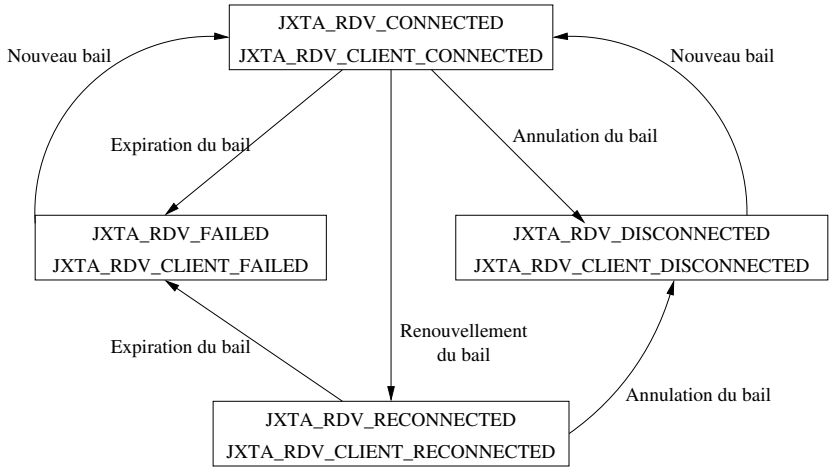

FIG. 6.5 – Diagramme des événements émis localement par le service de rendez-vous en fonction des étapes de gestion d'un bail. Chaque case contient les événements sur un pair standard (haut) et sur un pair rendez-vous (bas).

vous offrent ce qui est appelé des baux (en anglais *lease*) qui permettent aux pairs standards de recevoir les messages de propagation émis au sein du groupe associé. Un pair standard émet une requête de bail à un pair rendez-vous, et ce dernier répond par un message d'acceptation pour ce nouveau bail. Ce dernier message comprend la durée du bail offert au pair standard, et cela en millisecondes. La figure 6.5 présente le diagramme des événements émis par le service rendez-vous sur un pair en fonction des étapes de gestion d'un bail. L'utilisation de la primitive add/remove_event_listener permet d'associer une fonction de rappel (en anglais *callback*), qui sera appelée lors de tout événement émis par le service rendez-vous concernant la gestion des baux. Par ailleurs, cette primitive permet d'être informé de tout changement dans l'état local du pair : si le pair local devient un pair rendez-vous ou standard.

Enfin, un mécanisme *d'auto-promotion* est utilisé pour maintenir un certain nombre de pairs rendez-vous dans un groupe. Il consiste à surveiller le nombre de pairs rendez-vous disponibles dans la vue locale des pairs. Si ce nombre descend en-dessous du nombre fixé pour ce groupe, au bout d'un certain temps aléatoire mais borné un pair standard va devenir pair rendez-vous. De manière similaire, si le nombre de pairs rendez-vous est trop important, au bout d'un certain temps aléatoire mais borné un pair rendez-vous va devenir pair standard. Dans les deux cas, la décision de changement de statut est prise localement par chaque pair rendez-vous, en fonction de la taille de leur vue locale.

Protocole de propagation. Ce protocole permet aux pairs d'un groupe de contrôler la propagation des messages. Il permet de déterminer la source du message, les pairs qui ont déjà été utilisés pour la transmission de la requête, mais également le nombre maximum de sauts logiques que la requête peut faire avant d'être détruite.

6.2.2 Service de découverte

Objectif. Comme nous l'avons vu précédemment, dans un réseau JXTA, chaque ressource disponible est décrite sous la forme d'une annonce. L'objectif du protocole de découverte est de permettre aux pairs de trouver de telles annonces, toujours au sein d'un groupe. Ce protocole utilise le protocole de résolution pour la propagation des requêtes sur le réseau de pairs rendez-vous. En guise de réponse, le pair émetteur peut recevoir zéro ou plusieurs messages qui contiennent les éventuelles annonces répondant aux critères de la requête.

Algorithmique. Les implémentations de JXTA utilisent une table de hachage dite faiblement cohérente (en anglais *Loosely-Consistent DHT*) pour la mise en œuvre de l'algorithmique du service de découverte. Dans la suite de ce manuscrit nous utilisons le terme LC-DHT pour désigner une telle table de hachage. L'algorithmique utilisée pour le fonctionnement de la LC-DHT est détaillée à l'annexe A.2.2.

Complexité. Les DHT classiques ont une complexité en $O(\log\ n)$[1] pour la publication, alors que la LC-DHT a une complexité en $O(1)$ pour cette même opération (2 messages dans le pire des cas). Toutefois pour la recherche, une approche basée sur une LC-DHT n'offre pas une garantie équivalente, en $O(\log\ n)$, à celle des autres systèmes à base de DHT. En effet, les implémentations actuelles de JXTA offrent une garantie de recherche en $O(r)$, où r est le nombre de pairs rendez-vous. En effet, le pire cas correspond à une ressource située sur un pair rendez-vous à l'opposé dans l'espace de hachage du pair rendez-vous contacté sur l'anneau. Le nombre de messages alors émis est en $O(r)$. Toutefois, s'il y a convergence des vues locales des r pairs rendez-vous, elle est en $O(1)$ (4 messages dans le pire des cas). En cas d'absence de cette convergence les paramètres de contrôle du mécanisme du parcours de l'anneau des pairs rendez-vous permettent en pratique de grandement la réduire. En revanche, cette approche évite d'avoir à maintenir les tables d'une DHT en évitant un trafic fréquent, coûteux et d'une latence relativement élevée. En contrepartie, il est possible d'obtenir des résultats incomplets dans le cas de vues locales très incohérentes. C'est la reconnaissance et la prise en compte de la nature très souvent volatile d'un réseau P2P qui a poussé les concepteurs de JXTA à fournir cette LC-DHT.

6.2.3 Service de canal virtuel

Dans cette section, nous décrivons le service de canal virtuel, pièce centrale pour la communication entre les pairs dans la spécification JXTA. Le service de canal virtuel repose sur le service point-à-point (en anglais *endpoint service*). Ce dernier a pour rôle d'abstraire les protocoles réseaux disponibles sur un pair, tels que TCP ou HTTP. Une présentation plus détaillée de la pile des protocoles de communication au sein de JXTA fait l'objet de la section 10.1.

Objectif. Comme son nom l'indique, le service de canaul virtuel permet de s'abstraire des canaux de communications utilisés pour envoyer des messages. Une telle abstraction est appelée un canal virtuel (voir définition 6.4). Le service de canal virtuel fournit l'illusion d'une

[1] En terme de nombre de sauts et avec n le nombre de pairs dans le système.

adresse virtuelle indépendante de la localisation des pairs dans le réseau, mais également de la topologie logique (en cas d'absence de chemin direct). Cette indépendance permet de cacher au développeur d'applications la dynamicité sous-jacente des pairs : au cours d'une série de messages échangés, les pairs peuvent migrer d'une adresse physique à une autre de manière transparente.

Interface externe. À titre d'illustration, la figure 6.1 présente le code nécessaire pour la résolution d'un canal virtuel et l'envoi d'un message sur le canal virtuel en sortie ainsi créé. L'implémentation utilisée est JXTA-C. Dans un premier temps, la bibliothèque est démarrée puis une référence sur le service de canaux virtuels est récupérée (lignes 5 et 6). Enfin, une fonction synchrone pour la résolution du canal virtuel est utilisée (ligne 7). Le canal virtuel en sortie ainsi créé est alors obtenu (ligne 8). Ce dernier est ensuite utilisé pour envoyer un message (ligne 10 à 16). Une autre solution consiste à utiliser la primitive asynchrone pipe_service_connect couplée avec une fonction de rappel.

Listing 6.1 – Code nécessaire pour l'envoi d'un message sur le canal virtuel résolu d'une manière synchrone.

```
1  Jxta_PG              *pg = NULL;
2  Jxta_pipe_service    *pipe_service = NULL;
3  Jxta_outputpipe      *outputpipe = NULL;
4
5  jxta_PG_new_netpg(&pg);
6  jxta_PG_get_pipe_service(pg, &pipe_service);
7  jxta_pipe_service_timed_connect(pipe_service, pipe_adv, 10000, NULL, &pipe);
8  jxta_pipe_get_outputpipe(pipe, &outputpipe);
9
10 Jxta_message         *msg = jxta_message_new();
11 char                 *payload = malloc(current_msg_size * sizeof(char));
12 payload = (char *) memset((void *) payload, 0x61, current_msg_size);
13 Jxta_message_element *el = jxta_message_element_new_2(ns, name, NULL, payload,
14                                                        current_msg_size, NULL);
15 jxta_message_add_element(msg, el);
16 jxta_outputpipe_send(output, message);
```

L'interface de programmation pour envoyer un message sur un canal virtuel de type unidirectionnel ou de type diffusion est identique : seule l'annonce décrivant le canal virtuel change. Enfin, pour la réception d'un message, le mécanisme est similaire : une fonction de rappel est associée à un canal virtuel en entrée créé par l'intermédiaire de la primitive pipe_service_timed_accept.

Algorithmique. L'identifiant du canal virtuel permet au protocole de liaison de canal virtuel de le résoudre, en y associant une adresse réseau valide, et cela avant que les pairs ne s'échangent des messages sur ce canal virtuel. Une telle résolution est alors vérifiée périodiquement, par exemple et par défaut toutes les 20 minutes dans JXTA-J2SE, et éventuellement résolue à nouveau si besoin. L'algorithme interne de ce protocole est détaillé dans l'annexe A.2.3.

Complexité. Les pairs qui ont créé un canal virtuel en entrée publient cette information via le service de découverte. La complexité de résolution d'un canal virtuel unidirection-

nel est donc équivalente à celle de la recherche d'une annonce via le service de découverte (voir section 6.2.2). Pour un canal virtuel de propagation, aucune résolution n'est réalisée. L'ensemble de pairs rendez-vous sont contactés et la complexité est donc $O(\frac{n}{2})$.

6.3 Discussion et conclusion

Nous avons tout d'abord présenté dans ce chapitre la spécification JXTA, notamment les protocoles et les notions qu'elle définit. Puis nous avons détaillé le fonctionnement de trois services au cœur des implémentations de référence de JXTA (JXTA-C et JXTA-J2SE) : le service de rendez-vous, le service de découverte et le service de canaux virtuels. Ce choix est motivé par leur utilisation dans l'implémentation de JUXMEM qui est présentée dans le chapitre 7. Par ailleurs, le chapitre 10 est consacré à l'évaluation des performances des couches de communication de JXTA-C et JXTA-J2SE, notamment au-dessus des réseaux haute performance disponibles sur une grille. Le chapitre 11 évalue à grande échelle le protocole de gestion de vue de JXTA-C du service de rendez-vous.

La spécification JXTA a pour objectif de fournir un environnement P2P générique, spécialisable selon les besoins des applications, indépendant des langages et des réseaux et permettant l'interopérabilité des systèmes P2P. Pour ce faire, elle définit six protocoles qui travaillent ensemble pour la découverte, l'organisation, la surveillance et les communications entre les pairs. En outre, elle définit les concepts de *pairs*, *groupes de pairs*, *annonces* décrivant les ressources disponibles sur un réseau, *canal de communication virtuel* et *service réseau*. L'éventail des fonctionnalités offertes est large et nous n'en avons couvert qu'une partie limitée, mais se situant au cœur de l'environnement. C'est un environnement qui dispose d'implémentations matures : JXTA-C et JXTA-J2SE. La communauté de développeurs et le nombre d'applications basées sur JXTA ne cessent d'augmenter permettant une évolution de plus en plus rapide de celle-ci. Ainsi, à notre connaissance l'environnement JXTA est l'environnement le plus avancé dans son domaine.

La spécification JXTA est très générale compte tenu de sa volonté d'être générique, mais également précise notamment sur les formats des protocoles. Toutefois, certains points restent flous. Ainsi, les groupes *world* et *net* sont introduits dans la spécification sans être définis. Par ailleurs, la spécification est constituée de deux parties : le cœur de la spécification que doit impérativement fournir tout implémentation (en anglais JXTA *core specifications*), et les services standards optionnels (en anglais JXTA *standard services*). Toutefois, cette seconde partie n'introduit que des protocoles et des représentations réseau des messages JXTA. Ainsi, l'emploi du terme service semble peu approprié et porte à confusion. En outre, la spécification n'indique pas de dépendance du protocole de liaison de canal virtuel sur le protocole de résolution, dépendance pourtant conceptuellement présente pour la résolution d'un canal virtuel. Le terme *endpoint protocol*, utilisé à plusieurs reprises dans les spécifications, n'est pas défini : s'agit-il du protocole implémenté par le service point-à-point ou du protocole de routage ? Enfin, le protocole de résolution que doit impérativement implémenter toute bibliothèque respectant les spécifications JXTA repose sur le protocole de rendez-vous d'après ces dernières. Toutefois, le protocole de rendez-vous fait partie des protocoles optionnels que doit fournir un pair JXTA.

Les implémentations JXTA-C et JXTA-J2SE comportent également des lacunes dans la définition et la formalisation de leurs comportements. Par exemple, une erreur couramment

faite par les nouveaux développeurs d'applications basées sur JXTA consiste à utiliser le service de découverte pour faire une recherche exhaustive. Le service de découverte n'a pas cet objectif, mais une telle recherche est implémentable en utilisant le protocole de résolution. Par ailleurs, l'interface de programmation de JXTA-J2SE ainsi que les nombreux choix de configuration rendent son utilisation compliquée au premier abord : un débutant se retrouve vite perdu. Enfin, des problèmes affectant la fiabilité et la performance de JXTA-J2SE tardent à être résolus, notamment au niveau de la couche de communication JXTA *sockets*. L'implémentation JXTA-C, elle, ne souffre pas de ces deux derniers défauts.

Chapitre 7

Implémentation de JUXMEM sur l'environnement pair-à-pair JXTA

Sommaire

Dans ce dernier chapitre de la deuxième partie du manuscrit, nous présentons la mise en œuvre de notre proposition d'architecture de service de partage de données pour grille décrite au chapitre 5. Une description de la spécification JXTA et de ses implémentations a été donnée dans le chapitre précédent.

Dans un premier temps, nous présentons une description de l'architecture générale en couches de notre implémentation et l'utilisation des services JXTA. Puis, dans un second temps, nous nous concentrons sur les fonctionnalités offertes par le cœur de cette implémentation : son noyau appelé *juk*, pour JUXMEM *micro-kernel*. Nous aborderons en particulier la gestion des ressources mémoire et des communications ainsi que la prise en compte de la volatilité dans un réseau virtuel JUXMEM. Enfin, avant de conclure sur une discussion de

l'implémentation de ce prototype, une micro-évaluation du surcoût de ce noyau et de ses performances est présentée.

7.1 Architecture générale

Dans cette section, nous décrivons la structuration générale de l'implémentation de notre proposition d'architecture JUXMEM. Nous énumérons les couches logicielles présentes dans son implémentation et nous détaillons leurs rôles. Enfin, nous nous attardons sur la couche la plus basse dans les implémentations de JUXMEM : le noyau *juk*.

Tout d'abord, éclaircissons l'existence des termes JUXMEM-C et JUXMEM-J2SE. Notre premier prototype implémentant l'architecture de JUXMEM, telle qu'elle a été définie au chapitre 5, repose sur la bibliothèque JXTA-J2SE et a donc été codé en Java. Cette implémentation est appelée JUXMEM-J2SE. Toutefois, en raison de l'utilisation de JUXMEM par des intergiciels principalement développés en utilisant les langages C/C++ et pour des raisons de performance (voir chapitre 10), nous avons progressivement basculé vers l'utilisation de la bibliothèque JXTA-C. Cette autre implémentation est appelée JUXMEM-C. Notons que ces implémentations sont compatibles entre elles. Ainsi, un réseau JUXMEM peut être composé d'une utilisation mixte des implémentations JUXMEM-C et JUXMEM-J2SE : un client JUXMEM-C peut dialoguer avec un fournisseur JUXMEM-J2SE par exemple.

Dans ce chapitre, nous axons notre présentation sur l'implémentation JUXMEM-C, version la plus aboutie du cœur de notre contribution : le noyau *juk*.

7.1.1 Structuration en couches

Les implémentations JUXMEM-C et JUXMEM-J2SE sont constituées de trois couches présentes sur l'ensemble des pairs : le noyau *juk*, la couche de tolérance aux fautes et enfin la couche des protocoles de cohérence. La figure 7.1 présente cette hiérarchie des couches du service JUXMEM, leurs rôles étant décrits ci-dessous, en partant du bas de la pile des couches.

Noyau *juk*. Le noyau *juk* est constitué de deux parties : 1) la partie haute disponible pour les applications utilisatrices de JUXMEM et 2) la partie basse disponible pour les couches des protocoles de cohérence et de tolérance aux fautes. L'objectif du noyau *juk*, appelé par la suite simplement *juk*, est en effet double. Premièrement, il s'agit de fournir aux applications une implémentation des spécifications de l'interface de programmation du service de partage de données. C'est l'objectif de la partie haute de *juk*. Deuxièment, il s'agit de mettre à disposition des briques de bases adaptées concernant la publication et la découverte de ressources, les communications ainsi que le stockage des données dans l'espace d'adressage des pairs. En outre, ces briques de bases doivent s'abstraire de l'environnement pair-à-pair (P2P) choisi pour l'implémentation de *juk*. Cette indépendance des couches supérieures vis-à-vis du choix de la bibliothèque P2P est précisément l'un des objectifs de la partie basse de *juk*.

Couche de tolérance aux fautes. L'objectif de cette couche est de tolérer les pannes franches (ou *crashs*) et les fautes par omission (perte de messages) qui peuvent se produire au niveau des entités intervenant dans la gestion de la cohérence d'un bloc de données. Cette couche de tolérance aux fautes repose sur l'utilisation de mécanismes classiques de l'algorithmique des systèmes distribués tels que la gestion des membres d'un

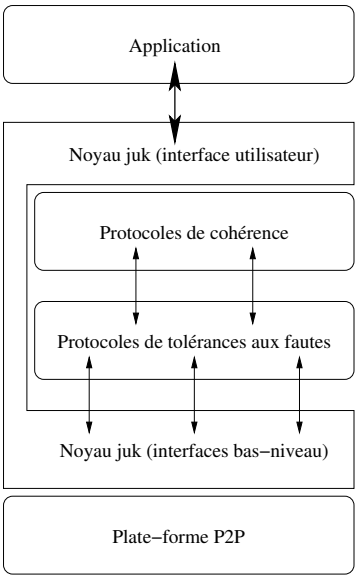

FIG. 7.1 – Hiérarchie des couches de l'architecture JUXMEM.

groupe, la diffusion atomique, le consensus et des détecteurs de fautes. Par exemple, le nœud qui héberge la copie de référence d'un bloc de données va être répliqué sur un ensemble de fournisseurs afin de tolérer sa défaillance. Mais, cette couche utilise également un mécanisme original de gestion de groupe auto-organisant hiérarchique qui permet de maintenir un degré de réplication suffisant par site d'une grille de calcul. L'objectif est toujours de conserver la donnée accessible pour les clients JUXMEM, et cela malgré les fautes qui peuvent se produire. Cette couche s'appuie sur *juk* pour ses besoins de communication mais également de découverte, par exemple des autres fournisseurs. Ainsi, dans le cas de l'exemple précédent, elle va notamment demander à *juk* de lui fournir de nouveaux fournisseurs susceptibles d'accueillir une copie du bloc de données.

Couche de protocoles de cohérence. L'objectif de cette couche est de s'assurer du partage cohérent d'une donnée entre un ensemble de clients. Pour ce faire, elle fournit les primitives de base pour l'implémentation de protocoles de cohérence définis dans le cadre des travaux menés sur les systèmes à MVP. Actuellement, cette couche offre un protocole hiérarchique et tolérant aux fautes qui implémente le modèle de la cohérence à l'entrée. Cette couche s'appuie sur la couche de tolérance aux fautes pour répliquer les entités critiques, traditionnellement présentes dans les protocoles issus des systèmes à MVP. En outre, la couche des protocoles de cohérence interagit avec *juk*, pour stocker dans l'espace d'adressage des fournisseurs les données partagées, mais également pour la gestion de l'hétérogénéité des architectures des machines sur lesquelles s'exécutent les pairs.

La gestion hiérarchique et conjointe des protocoles de cohérence d'un côté, et les mécanismes de tolérance aux fautes de l'autre, font l'objet des travaux de thèse de Sébastien Monnet. De ce fait, ces deux couches ne seront pas plus détaillées dans ce manuscrit. Le lecteur intéressé peut se reporter à [127].

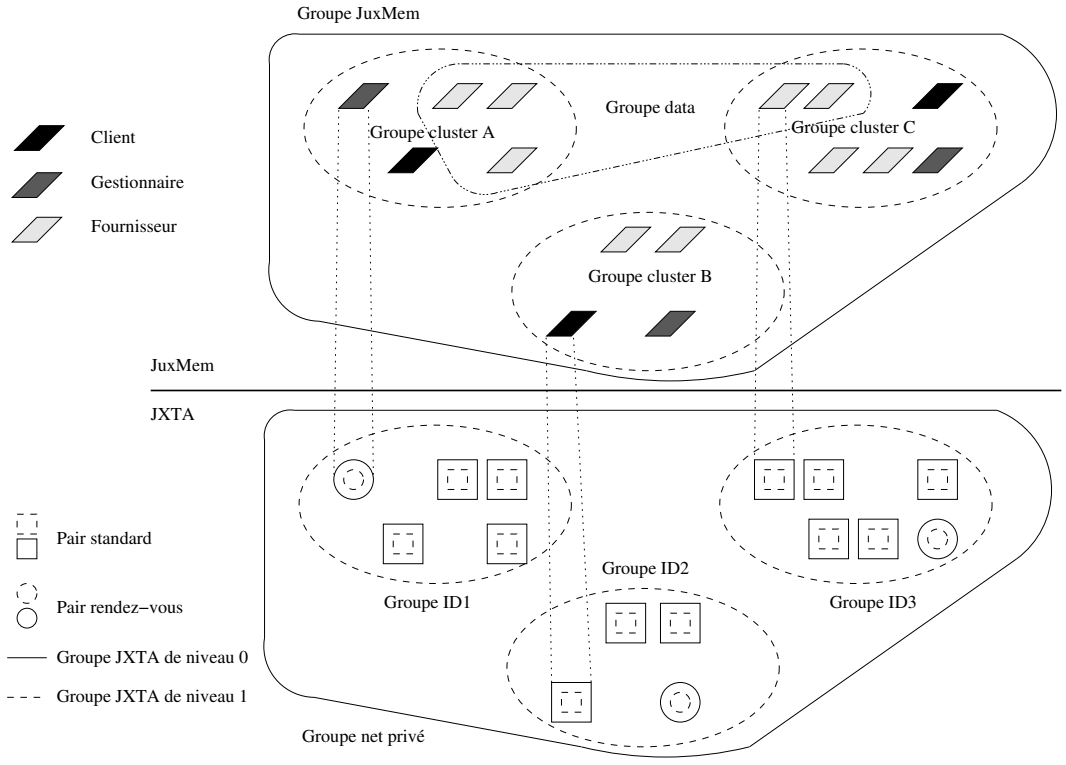

FIG. 7.2 – Projection des entités JUXMEM sur les entités JXTA.

Les deux implémentations JUXMEM-C et JUXMEM-J2SE représentent respectivement 13 500 et 16 700 lignes de code. Elles reposent respectivement sur JXTA-C 2.3 et JXTA-J2SE 2.3.5. La dernière version de JUXMEM (0.3) date du 7 août 2006.

7.1.2 Présentation générale de l'utilisation des concepts de JXTA

Avant de présenter le cœur de notre contribution à l'implémentation de notre service de partage de données et d'en détailler le fonctionnement interne. Nous introduisons notre utilisation des concepts de JXTA pour l'implémentation de JUXMEM. Cette correspondance est valable quelle que soit l'implémentation de JUXMEM choisie.

La figure 7.2 reprend le réseau JUXMEM de la figure 5.4 et présente notre utilisation des groupes JXTA ainsi que la projection des pairs JUXMEM sur les types de pairs JXTA existants. Nous commençons tout d'abord par expliquer les groupes et les services JXTA qui sont présents dans JUXMEM. Puis, nous présentons la correspondance entre les pairs JUXMEM et les pairs JXTA au sein de chaque groupe.

Groupes et services JXTA. Les notions de groupes de pairs et de services réseau de JXTA (voir chapitre 6) correspondent à nos besoins pour l'implémentation du concept de service de partage de données. En effet, un service réseau JXTA de groupe est rendu par un ensemble de pairs pouvant potentiellement collaborer pour y parvenir (voir définition 6.5). Les implémentations JUXMEM-C et JUXMEM-J2SE sont donc des services

réseau de groupe, au sens JXTA du terme. Ainsi, le groupe JUXMEM est un groupe JXTA qui définit, outre les services habituels d'un groupe JXTA (service de découverte, de canal virtuel, de rendez-vous, etc.), le service JUXMEM. Ce dernier respecte l'interface de programmation mise à disposition d'un développeur de services JXTA, notamment pour la gestion de son cycle de vie. Toutefois et comme indiqué sur la figure 7.2, le groupe JUXMEM est un groupe *net*, c'est-à-dire le groupe dans lequel tous les pairs sont lancés (voir section 6.1.2). Un tel groupe est noté groupe de niveau 0 sur la légende de la figure. Toutefois ce groupe est privé : il utilise son propre identifiant et les pairs membres de ce groupe sont donc isolés des autres pairs JXTA s'exécutant dans leur voisinage accessible. Notons que plusieurs identifiants différents peuvent être utilisés, formant ainsi plusieurs réseaux JUXMEM différents et privés. Enfin, les groupes CLUSTER définis dans l'architecture JUXMEM sont également des groupes JXTA. Ils ne définissent pas de services particuliers et sont inclus dans le groupe JUXMEM, et donc notés groupes de niveau 1. Par ailleurs leur annonce est publiée dans le groupe JUXMEM afin que d'autres pairs, s'exécutant sur la même grappe de machines par exemple, puissent y accéder. La figure 7.2 présente trois instances de ces groupes CLUSTER, chacun disposant de leur propre identifiant $ID1$, $ID2$ et $ID3$.

Pairs JXTA. Comme nous l'avons vu au chapitre 6, les implémentations de la spécification JXTA choisissent d'organiser leur réseau de pairs en utilisant des pairs rendez-vous en guise de super-pairs (voir section 6.2.1). Les pairs standards sont en effet abonnés à des pairs rendez-vous qui jouent le rôle de super-pairs. Cette structure du réseau logique correspond à nos besoins, notamment pour l'implémentation de notre algorithme d'allocation mémoire (voir section 5.2.3.1). Comme le montre la figure 7.2, les pairs fournisseurs et clients de JUXMEM correspondent à des pairs JXTA standards dans le groupe JUXMEM mais aussi dans le groupe CLUSTER dont ils dépendent. Un pair JUXMEM de ce type est schématisé par un double rond sur la figure, afin de symboliser cette projection en pair standard dans les deux groupes à la fois. Les pairs gestionnaires sont eux des pairs rendez-vous également à la fois dans le groupe JUXMEM et leur groupe CLUSTER. En effet, l'utilisation d'un groupe *net* privé nécessite la mise en place de pairs rendez-vous dédiés. En contrepartie, un meilleur contrôle de la topologie du réseau virtuel P2P et une meilleure adéquation à la topologie physique sous-jacente sont possibles. Cela se traduit par des coûts réduits d'opérations, tels que ceux de la découverte de ressources. Les pairs fournisseurs et clients sont donc interconnectés à leurs pairs gestionnaires par deux baux : un dans le groupe JUXMEM et un dans le groupe CLUSTER dont ils dépendent.

7.1.3 Le cœur de JUXMEM : le noyau *juk*

Dans cette section, nous introduisons le cœur de notre contribution à l'implémentation de l'architecture JUXMEM : son noyau appelé *juk*. L'objectif de *juk* est double :

1. implémenter la spécification d'un service de partage de données ;
2. fournir aux couches de protocoles de cohérence et de tolérance aux fautes une interface de programmation indépendante de la bibliothèque P2P utilisée.

Dans JUXMEM-C, son implémentation la plus aboutie, *juk* représente 7 500 lignes de code.

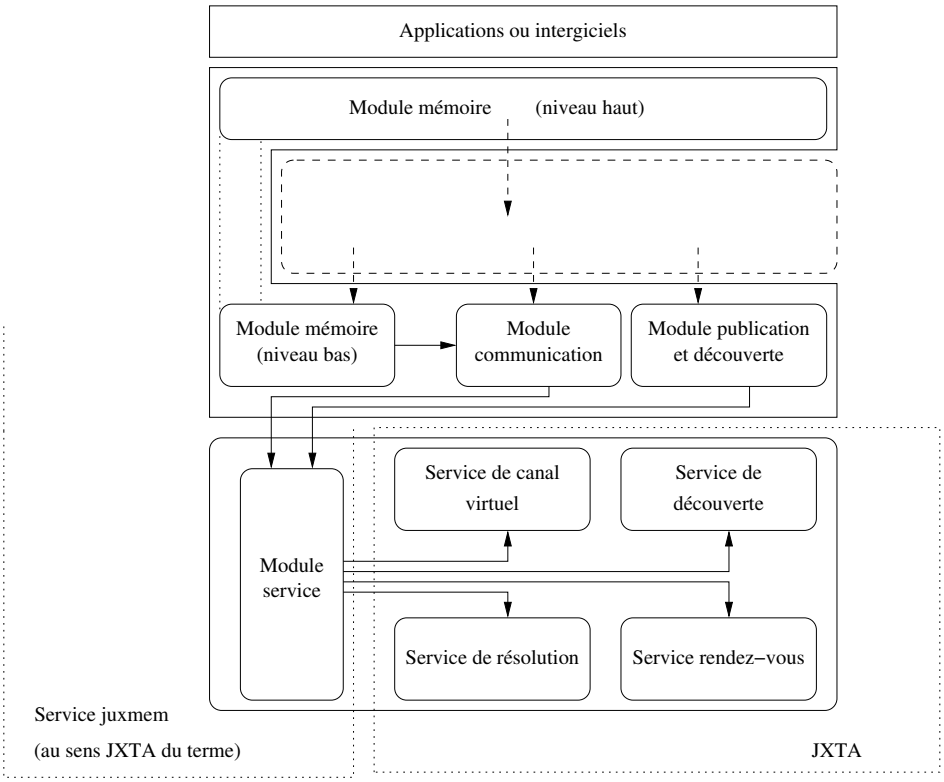

FIG. 7.3 – Structuration en modules et dépendances envers les spécifications JXTA du noyau de l'architecture JUXMEM appelé *juk*.

La figure 7.3 présente les différents modules qui composent les deux parties de *juk* ainsi que leurs interactions. La partie basse de *juk* est ainsi constituée d'un module mémoire, d'un module de communication entre pairs, d'un module de publication et de découverte de ressources et enfin d'un module service. Le module mémoire est cependant scindé en deux afin de constituer la partie haute de *juk* qui est utilisée par les développeurs d'applications ou d'intergiciels pour le partage de données. Les autres modules sont utilisés aussi bien, et selon leurs besoins, par la couche de tolérance aux fautes que par la couche des protocoles de cohérence. La partie basse du module mémoire dépend en partie du module de communication. Enfin, tous les modules de la partie basse de *juk* reposent sur le module service. Le fonctionnement interne de l'ensemble de ces modules est détaillé dans la suite de ce chapitre.

Dans son implémentation actuelle, la partie basse de *juk* repose sur la spécification JXTA. Ainsi, la figure 7.3 représente également les dépendances de *juk* envers la spécification JXTA. Plus précisément, l'implémentation actuelle du module communication repose sur le service de canal virtuel de JXTA, l'implémentation du module publication et découverte se fonde sur le service de découverte JXTA. Le module mémoire, quant à lui, utilise le service de résolution et le service de rendez-vous de JXTA. Toutes ces dépendances passent par le module service. Enfin, le module service lui-même repose sur le service de rendez-vous. Le fonctionnement de ces services JXTA a fait l'objet d'une présentation au chapitre 6.

Le deuxième objectif de *juk*, c'est-à-dire s'abstraire des interfaces de programmation de la bibliothèque P2P utilisée, part du constat que JXTA ne spécifie que les protocoles. Ainsi, l'interface de programmation d'un service JXTA ainsi que sa mise en œuvre peuvent être différentes d'une bibliothèque implémentant la spécification JXTA à une autre. En outre, il est nécessaire de pouvoir modifier l'implémentation d'un ou plusieurs modules de *juk* sans devoir adapter les couches supérieures utilisatrices. Par exemple, le module de communication peut être implémenté en se basant sur d'autres protocoles JXTA comme le service de communication point-à-point. Les modifications internes ne doivent pas alors avoir d'impact sur les modules dépendant du module communication. Pour atteindre cet objectif d'indépendance vis-à-vis de l'implémentation JXTA choisie, l'interface de programmation externe de *juk* offerte aux développeurs des couches supérieures, a été précisément définie. Par ailleurs, cela offre la possibilité d'implémenter *juk* sur d'autres bibliothèques P2P telles que FreePastry, Bamboo, voire même sur une implémentation de JXTA intégrant les algorithmes utilisés par ces bibliothèques [178].

Comme le souligne la figure 7.3, le module service joue un rôle central dans *juk*. Ce module s'occupe de l'insertion de *juk* dans la bibliothèque P2P utilisée en respectant, par exemple, son modèle de programmation pour la gestion du cycle de vie des services. Dans notre cas, la bibliothèque JXTA-C nécessite l'ajout de la bibliothèque JUXMEM en tant que service supplémentaire dans un groupe *net* privé. La bibliothèque JUXMEM sera alors dynamiquement chargée à l'exécution lors de la création du groupe *net* privé et de ses services. Le service JUXMEM, au sens JXTA du terme, sera alors créé. Celui-ci va construire durant cette phase d'initialisation un objet central dans l'implémentation de *juk* : la structure JuxMem_peer. Un pointeur sur cette structure est récupéré via la routine juxmem_get_peer. Toujours au sens JXTA du terme, cette routine constitue l'unique interface du service JUXMEM. Elle est accessible depuis une référence sur le groupe JUXMEM, via l'identifiant du service JUXMEM. Elle va en outre permettre d'accéder à l'interface de programmation du service de partage de données telle qu'elle a été spécifiée dans le chapitre 5. Notons toutefois que ce mécanisme est transparent pour l'utilisateur de JUXMEM-C : un simple appel à la routine juxmem_initialize est suffisant. En effet, cette routine s'occupe du lancement du groupe *net* JUXMEM, de la récupération du pointeur sur l'interface de programmation du service JUXMEM ainsi que de l'appel à la routine juxmem_get_peer. Cette dernière routine retourne un pointeur sur une structure de type JuxMem_peer, qui est nécessaire pour utiliser les opérations de l'interface de programmation de JUXMEM. Enfin, cette structure contient un pointeur sur le service JUXMEM, des informations sur le pair comme son nom et son identifiant ainsi que sur les différents modules qui constituent *juk*, nécessaires au bon fonctionnement du service JUXMEM.

7.2 Gestion de la volatilité

La structure JuxMem_peer dispose également de pointeurs sur des objets de type JuxMem_client, JuxMem_provider et JuxMem_manager. Ces structures correspondent aux différentes *personnalités* que peut prendre (de manière simultanée ou non) un pair JUXMEM. Dans cette section, nous décrivons les différents changements de personnalité que peuvent réaliser les pairs JUXMEM, en fonction des conditions de volatilité dans l'infrastructure physique sous-jacente.

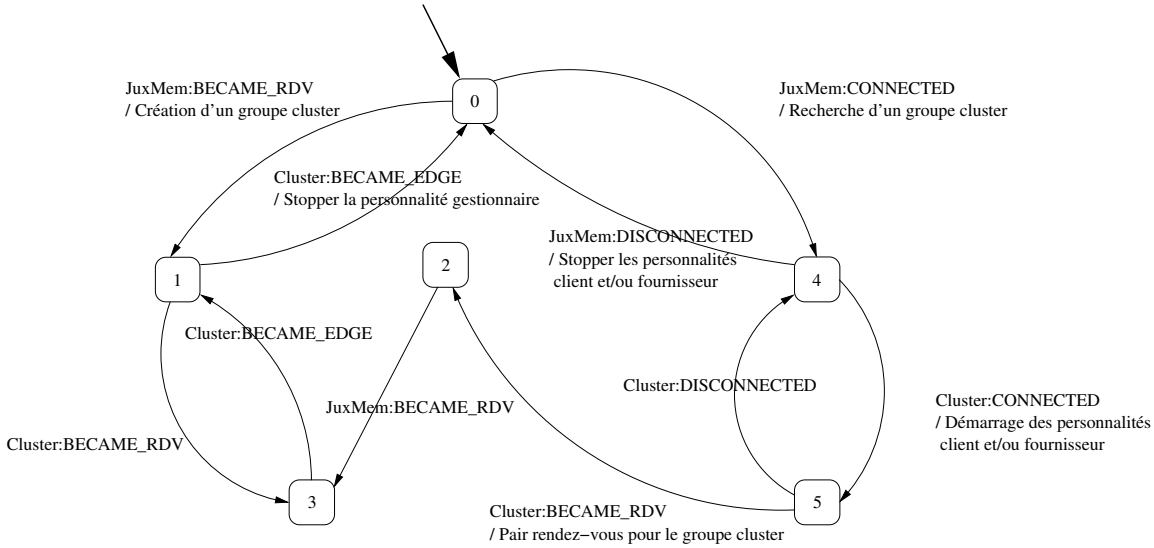

FIG. 7.4 – Automate états actions pour la gestion de la volatilité dans JUXMEM.

La volatilité dans un réseau P2P se caractérise par la perte ou l'ajout de pairs dans le système, avec en conséquence une éventuelle réorganisation du réseau logique. Dans le cadre d'un réseau JUXMEM cela peut se traduire par les cas suivants :

1. la perte de gestionnaires et/ou de fournisseurs ;
2. l'ajout de fournisseurs, de clients et/ou de fournisseurs ;
3. le démarrage des personnalités gestionnaire et/ou fournisseur ;
4. l'arrêt des personnalités gestionnaire et/ou fournisseur.

Nous tenons à préciser que la perte de clients, et donc de processus applicatifs, sont considérés comme étant à la charge de l'application. Au niveau JXTA, la volatilité peut se traduire par :

1. l'ajout et/ou le retrait de pairs standards et/ou de pairs rendez-vous ;
2. la transformation de pairs rendez-vous en pairs standard et inversement[1].

Compte tenu de l'utilisation d'une hiérarchie de groupes à deux niveaux dans JUXMEM, de tels changements peuvent avoir lieu dans le groupe JUXMEM mais également dans le groupe CLUSTER. Afin de réagir aux événements de réorganisation des groupes JUXMEM et CLUSTER, le module service de *juk* enregistre une fonction de rappel à la fois au niveau du service rendez-vous du groupe JUXMEM mais également au niveau du groupe CLUSTER. À partir des événements émis dans chaque groupe par le pair local (voir figure 6.5), nous pouvons établir l'automate des états d'un pair JUXMEM. L'objectif de cet automate est de prendre en compte les informations de volatilité fournies par JXTA afin d'effectuer, dans chaque cas, les actions nécessaires au niveau JUXMEM. La figure 7.4 représente ces actions associées à chaque transition d'un état à un autre de cet automate.

L'état d'un pair JUXMEM dans cet automate définit le rôle du processus dans sa participation au service JUXMEM : fournisseur, client ou gestionnaire. Un fournisseur est défini

[1]Ceci est réalisé par le mécanisme d'auto-promotion du service rendez-vous de JXTA (voir section 6.2.1).

comme un pair standard à la fois dans le groupe JUXMEM et le groupe CLUSTER, c'est-à-dire dans l'état 5 sur la figure. Cet état définit également un client, la seule différence étant qu'il ne met à la disposition du service aucun bloc mémoire. Enfin, un gestionnaire est défini comme un pair dans l'état rendez-vous, à la fois dans le groupe JUXMEM et dans le groupe CLUSTER. Un tel pair JUXMEM se trouve donc dans l'état 3 sur la figure. Lorsque qu'un pair joue plusieurs rôles de manière simultanée, le rôle de gestionnaire l'emporte par rapport au rôle de client et/ou de fournisseur. Plus précisément, un pair exécutant à la fois le rôle de gestionnaire et de fournisseur par exemple est un pair rendez-vous au niveau JXTA dans les groupes JUXMEM et CLUSTER.

Nous illustrons le fonctionnement de cet automate à l'aide de deux scénarios. Le premier concerne le démarrage d'un pair JUXMEM en tant que gestionnaire. Il est composé des étapes suivantes.

1. Le pair passe de l'état 0 à 1 lorsque tous les services JXTA ont activé leur rôle de pair rendez-vous dans le groupe JUXMEM. Cela se traduit par l'émission de l'événement JXTA_RDV_BECAME_RDV dans le groupe JUXMEM. À l'entrée de cet état au niveau JUXMEM, son rôle de gestionnaire dans le groupe JUXMEM est démarré. Il consiste principalement à créer un groupe CLUSTER, à le démarrer en tant que pair rendez-vous et à le publier dans le groupe JUXMEM.
2. Le pair passe de l'état 1 à 2 lorsque tous les services JXTA ont activé leur rôle de pair rendez-vous dans le groupe CLUSTER. Cela se traduit par l'émission de l'événement JXTA_RDV_BECAME_RDV dans le groupe CLUSTER. Aucune action particulière n'est réalisée à l'entrée dans cet état, si ce n'est la mise à jour de l'automate.

Le deuxième scénario concerne le démarrage d'un pair JUXMEM en tant que fournisseur. Il est composé des étapes présentées ci-dessous.

1. Le pair passe de l'état 0 à 4 lorsqu'il obtient un bail, dans le groupe JUXMEM, du pair rendez-vous avec lequel il a été configuré. Cela se traduit par l'émission de l'événement JXTA_RDV_BECAME_EDGE dans le groupe JUXMEM. À l'entrée de cet état au niveau JUXMEM, un groupe CLUSTER est recherché sur le gestionnaire auquel le fournisseur n'est pour l'instant que partiellement connecté. Lorsqu'un groupe CLUSTER est trouvé, il est instancié afin d'obtenir du gestionnaire un bail dans ce groupe
2. Le pair passe de l'état 4 à 5 lorsqu'il obtient un bail dans le groupe CLUSTER qu'il a précédemment instancié. Le fournisseur est alors connecté au gestionnaire à la fois dans le groupe JUXMEM et CLUSTER : il est considéré comme opérationnel. À l'entrée de cet état au niveau JUXMEM, le fournisseur va alors publier l'annonce de l'espace mémoire qu'il met à disposition du service.

La séquence qui fait passer un pair de l'état 5 à 2 puis 3 correspond à la promotion d'un fournisseur ou d'un client en gestionnaire suite à la perte d'un gestionnaire dans le réseau JUXMEM considéré. Au niveau JXTA, la promotion intervient tout d'abord dans le groupe CLUSTER auquel appartenait le gestionnaire défaillant. À l'entrée de l'étape 2, le code du service JUXMEM promeut le processus en tant que pair rendez-vous dans le groupe JUXMEM. Un pair passe de l'état 3 à l'état 5 lorsque un trop grand nombre de gestionnaires sont présents dans un réseau JUXMEM. Enfin, un gestionnaire, un client ou un fournisseur passe dans l'état 0 lorsque le processus dans lequel le pair s'exécute se termine.

7.3 Gestion des ressources mémoire

Dans cette section, nous décrivons tout d'abord le module mémoire. Il se décompose en deux sous-modules : l'un pour la découverte et l'allocation d'espaces mémoire, et l'autre pour le stockage de blocs de données. Puis dans un deuxième temps, nous présentons le module de publication et de découverte. Dans ces deux parties, nous présentons des exemples d'utilisation de ces modules par les couches des protocoles de cohérence et de tolérance aux fautes.

7.3.1 Découverte et allocation mémoire

Objectif et interface externe. L'objectif du module mémoire haut est de fournir l'interface de programmation de notre concept de service de partage de données. Cette interface a été spécifiée à la section 5.1.4. Les opérations permises par le module mémoire haut permettent, par exemple, d'allouer des espaces mémoire (juxmem_malloc), de projeter des espaces mémoire dans l'espace d'adressage des clients JUXMEM (juxem_mmap), d'accéder aux données de manière cohérente (voir section 5.1.4.2), etc. Cette interface de programmation est également disponible dans le langage C++.

Implémentation et exemple d'utilisation. Afin d'atteindre son objectif, le module mémoire haut interagit avec la couche des protocoles de cohérence, en particulier lors des accès aux données. Chaque client peut avoir besoin d'accéder à plusieurs données et peut projeter une même donnée dans différents endroits de son espace d'adressage. Via une table de hachage, un *descripteur de mémoire* est associé à chaque pointeur sur une donnée partagée. Le listing 7.1 représente l'ensemble des champs de cette structure (de type JuxMem_descriptor), qui inclut notamment l'identifiant de la donnée (ligne 6), les structures XDR (*eXternal Data Representation* [162]) nécessaires pour la gestion de l'hétérogénéité (lignes 8 à 11), la taille de donnée (ligne 5), etc. Le champ key, appelé *identifiant local de données*, autorise la projection d'une même donnée dans deux zones mémoire différentes au sein d'un même processus client, lorsque le mode zéro-copie pour le stockage des blocs de données est activé (voir section 7.3.2 pour plus de détails). Cet identifiant local de données est constitué de la concaténation de l'identifiant de la donnée correspondante, avec la valeur du pointeur de la zone mémoire utilisée localement pour stocker cette donnée.

Listing 7.1 – Champs de la structure JuxMem_descriptor (version simplifiée).

```
1  struct _juxmem_descriptor {
2    consistency_protocol *coherency_protocol;
3
4    void  *ptr;
5    size_t size;
6    char  *data_id;
7    char  *key;
8
9    char *xdr_data_ptr;
10   XDR  *xdr_encode_ptr;
11   XDR  *xdr_decode_ptr;
12   u_int xdr_size;
13 };
14 typedef struct _juxmem_descriptor JuxMem_descriptor;
```

```
<?xml version=\"1.0\"?>
<!DOCTYPE JuxMem:ProviderAdvertisement>
<JuxMem:ProviderAdvertisement>
<Id>urn:jxta:uuid-B049BE83E5B9B9CD19570204</Id>
<Type>JxtaUnicast</Type>
<Name>Fournisseur10</Name>
<Memory range=(0 :: 104448)>#1024</Memory>
</JuxMem:ProviderAdvertisement>
```

Figure 7.5 – Exemple d'annonce de type fournisseur.

C'est lors de l'utilisation des primitives d'allocation mémoire (juxmem_malloc et juxmem_realloc) ou de projection d'espaces mémoire (juxmem_mmap et juxmem_attach) que les structures JuxMem_descriptor associées à chaque donnée sont créées. Ce descripteur mémoire va permettre de déterminer, via l'identifiant de la donnée, l'objet implémentant le protocole de cohérence associé à la donnée (champ coherency_protocol, ligne 2 du listing 7.1). L'opération adéquate du protocole de cohérence peut alors être appelée.

Par ailleurs, les primitives d'allocation d'espace mémoire ont pour objectif de découvrir les espaces mémoire disponibles dans JUXMEM. Chaque fournisseur décrit l'espace mémoire dont il dispose par l'intermédiaire d'une annonce JXTA. Une telle annonce est appelée *annonce fournisseur*. La figure 7.5 en présente un exemple. Nous pouvons constater que ce type d'annonce spécialise une annonce de canal virtuel (voir figure 6.2), qui correspond au canal sur lequel le fournisseur écoute les demandes d'allocation. Par rapport à une annonce de canal virtuel, une annonce fournisseur comprend en plus le champ Memory. Il définit le nombre de blocs mémoire offert par un fournisseur (par défaut la taille d'un bloc est de 1024 octets). Chaque fournisseur envoie une copie de son annonce au gestionnaire dont il dépend, en utilisant le module de publication et découverte de *juk* (voir section 7.3.3).

Le processus d'allocation mémoire n'utilise toutefois pas le module de découverte de *juk*. Il repose directement sur les services de résolution et de rendez-vous de JXTA, et cela au niveau du groupe JUXMEM. Sur la base des informations de la vue locale des gestionnaires dans ce groupe, ce choix d'implémentation a pour avantage de permettre d'orienter les requêtes d'allocation vers les gestionnaires disposant d'espaces mémoire suffisants. Une description plus détaillée du fonctionnement de cet algorithme a été donnée dans la section 5.2.3.1. Ainsi, par l'intermédiaire du service de résolution de JXTA et au niveau du groupe JUXMEM, chaque gestionnaire enregistre une fonction qui va être appelée lors de la réception d'une requête d'allocation. La figure 7.6 présente le diagramme d'action de la routine de réception d'une telle requête. Notons m le degré demandé de réplication par site, n le nombre de sites demandées et x la taille de la donnée. L'algorithme est le suivant.

1. Chaque gestionnaire vérifie si la requête ne concerne pas une demande d'allocation dans son groupe CLUSTER. Dans ce cas, il transmet la requête aux n « meilleurs » gestionnaires de la vue locale du groupe JUXMEM (voir section 5.2.3.1 pour plus de précision sur « meilleur ») et l'algorithme se termine. Sinon, l'algorithme passe à l'étape suivante.
2. Le gestionnaire vérifie si n est supérieur à 1. Si c'est le cas, il transmet alors la requête d'allocation aux $n - 1$ « meilleurs » gestionnaires de sa vue locale.
3. Enfin, le gestionnaire inspecte son cache local des annonces fournisseur pour en ex-

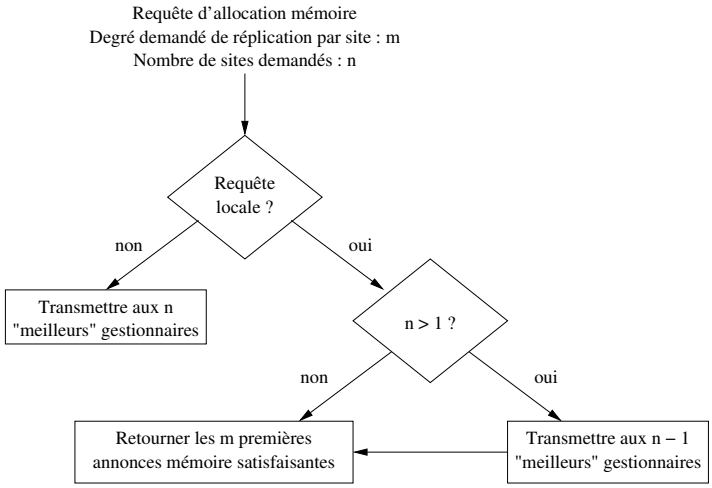

FIG. 7.6 – Diagramme d'actions de la routine de réception d'une requête d'allocation.

traire au maximum m annonces mémoires. Cette liste est retournée au client. La taille de l'espace mémoire disponible sur chaque fournisseur, dont l'annonce a été sélectionnée, est alors mis à jour dans le cache local du gestionnaire.

Le client peut alors à proprement parler demander l'allocation des espaces mémoire sur les différents fournisseurs dont il dispose. Pour ce faire, il va leur envoyer un message d'allocation en passant en paramètre au module communication de *juk* (voir section 7.4) l'identifiant du canal de communication contenu dans les annonces fournisseur.

L'existence d'une donnée stockée dans JUXMEM est rendue « publique », c'est-à-dire accessible pour d'autres clients, par une *annonce mémoire*. L'identifiant d'une annonce mémoire correspond à l'identifiant de la donnée. Les annonces mémoire disposent par ailleurs d'un champ contenant un identifiant, qui permet de dialoguer avec les fournisseurs membres du groupe DATA, associé à la donnée.

7.3.2 Stockage des blocs de données

Objectif. L'objectif du module mémoire bas est de gérer le stockage des espaces mémoire dans l'espace d'adressage des fournisseurs. Ce module est utilisé par la couche des protocoles de cohérence. Ainsi, ils n'ont pas à se soucier de l'organisation du stockage des données sur les fournisseurs ou à tenir compte de l'hétérogénéité des machines. Les protocoles de cohérence peuvent donc se concentrer sur les actions (règles) de cohérence à appliquer sur les espaces mémoire lors de l'utilisation, par les programmeurs d'applications, des différentes primitives disponibles dans le modèle d'accès mémoire (voir section 5.1.4.2). L'encodage des données est nécessaire afin de prendre en compte l'hétérogénéité des machines constituant une grille de calcul : architecture 32 ou 64 bits, représentation mémoire petit-boutiste (en anglais *little endian*) ou gros-boutiste (en anglais *big endian*). Par exemple, ce module s'occupe de l'encodage et du décodage des données lors de leurs transferts entre clients et fournisseurs par les protocole de cohérence. Pour cette gestion de l'hétérogénéité, JUXMEM utilise la bibliothèque XDR.

Interface externe et exemple d'utilisation. Du côté client, le module mémoire bas offre aux couches supérieures les primitives qui sont décrites ci-dessous.

juxmem_from_user_memory_to_message. Cette primitive permet d'ajouter une donnée de taille size octets pointée par ptr dans le message passé en paramètre. Elle est par exemple appelée par les protocoles de cohérence lors du relâchement du verrou, pris en mode exclusif et associé à la donnée. Le listing 7.2 représente cet exemple d'utilisation. Un message JXTA est crée (ligne 2), puis la donnée du client ajoutée dans le message par le biais de cette primitive (ligne 5), enfin le message est envoyé (ligne 8).

Listing 7.2 – Utilisation de la primitive juxmem_from_user_memory_to_message dans la couche des protocoles de cohérence, lors du relâchement d'un verrou exclusif (code simplifié).

```
1  int ec_release(consistency_protocol *self, void *ptr, size_t size) {
2    Jxta_message *msg = jxta_message_new();
3    ...
4    inc_version(self->impl->data_version);
5    juxmem_from_user_memory_to_message(self->impl->peer, self->impl->data_id,
6                                       ptr, size, msg);
7    ...
8    send_unlock_request(self->impl->comm, msg);
9    ...
10 }
```

juxmem_to_user_memory_from_message. Cette primitive permet d'extraire la donnée du message passé en paramètre pour mettre à jour l'espace mémoire du processus client pointé par ptr et d'une taille size octets. Elle est par exemple appelée par les protocoles de cohérence lors de la prise du verrou associé à la donnée.

Du côté fournisseur, le module mémoire bas offre les primitives décrites ci-dessous. Toutes ces primitives sont appelées par la couche des protocoles de cohérence.

juxmem_memory_malloc. Cette primitive permet d'allouer une collection de pages pour stocker une donnée de la taille passée en paramètre (en octets). La taille par défaut d'une page utilisée pour le stockage des données sur les fournisseurs est de 2048 ko (valeur paramétrable). Notons que si le support XDR est activé, la taille passée en paramètre est multipliée par 4 pour tenir compte du surcoût en termes d'espace mémoire pour stocker la donnée encodée. Cette primitive est appelée sur les fournisseurs lors du traitement de demande d'allocation mémoire.

juxmem_memory_free. Cette primitive permet de libérer la collection de pages allouées pour la donnée dont l'identifiant est passé en paramètre. Elle est appelée lorsque la donnée est détruite sur le fournisseur.

juxmem_to_memory_from_message. Cette primitive permet d'ajouter dans le message passé en paramètre le contenu de la donnée, dont l'identifiant est passé en paramètre. Elle est appelée pour mettre à jour la copie de la donnée sur le fournisseur.

juxmem_from_memory_to_message. Cette primitive permet d'extraire du message passé en paramètre le contenu de la donnée, dont l'identifiant est passé en paramètre. Elle est appelée pour envoyer une copie de la donnée du fournisseur vers le client.

Implémentation. Le listing 7.3 présente le code utilisé pour le décodage d'un bloc de données reçu dans un message du côté client. Le code présenté est celui exécuté lorsque l'option XDR est activée. Un des aspects qui guide la conception de JUXMEM est la performance de l'accès aux données. Ainsi, ce listing illustre le mécanisme utilisé pour minimiser le nombre de copies de la donnée lors de sa réception dans un message. Un mécanisme d'appel de fonctions préenregistrées disponible dans JXTA-C permet de copier directement une partie du message à l'adresse mémoire retournée par cette fonction (appelé mode zéro-copie). Sur la figure, cette fonction s'appelle memory_callback (ligne 1). Une description plus détaillée de ce mécanisme est donnée à la section 10.2.2. Dans l'exemple présenté, la fonction est appelée dès qu'un identifiant local de donnée enregistré est contenu dans le message[2] (voir section 7.3.1 pour la définition d'un identifiant local de données). Ainsi, l'adresse correcte du tampon XDR de la donnée est calculée, en tenant compte de l'offset, puis retournée à JXTA-C (lignes 9 à 17). Ce dernier utilise ce tampon pour stocker la donnée contenue dans le message. Enfin, lors de l'appel à la primitive juxmem_to_memory_from_message (ligne 21), la donnée est décodée via XDR dans l'espace mémoire où le client a alloué la donnée (ligne 33). Dans le cas où l'option XDR n'est pas activée, la donnée extraite du message est directement copiée à l'adresse mémoire utilisée par le client, via notre mécanisme de fonctions pré-enregistrées.

Listing 7.3 – Code utilisé pour le décodage d'une donnée reçue dans un message, du côté client.

```
1  static void* JXTA_STDCALL memory_callback (const char *ns,
2                                             const char *element_name,
3                                             const char *element_mime_type) {
4    JuxMem_descriptor *descriptor = NULL;
5    void              *user_buffer = NULL;
6    char              *key = malloc(32);
7
8    user_buffer = (void *) apr_hash_get(self_static->user_buffers,
9                                        (const void *) ns, APR_HASH_KEY_STRING);
10
11   sprintf(key, "%li", (long) user_buffer);
12   descriptor = (JuxMem_descriptor *) apr_hash_get(self_static->client_descriptors,
13                                                   (const void*) key,
14                                                   (apr_ssize_t) strlen(key));
15   free(key);
16   return ((char *) descriptor->xdr_data_ptr) +
17           (atoi(element_name) * JUXMEM_CHUNK_SIZE);
18 }
19
20 int juxmem_to_user_memory_from_message(JuxMem_peer *peer, Jxta_id *data_id,
21                                        void *ptr, size_t size,
22                                        Jxta_message *message) {
23   JuxMem_peer_memory *self = juxmem_peer_get_memory(peer);
24   JuxMem_descriptor  *descriptor = NULL;
25   u_int               xdr_size = 0;
26   char               *key = malloc(32);
27
28   sprintf(key, "%li", (long) ptr);
29   descriptor = (JuxMem_descriptor *) apr_hash_get(self->descriptors,
```

[2] Sur le listing 7.3, l'étape d'enregistrement de la fonction de rappel n'est pas représentée pour des raisons de lisibilité.

```
30                                                  (const void*) key,
31                                                  (apr_ssize_t) strlen(key));
32    xdr_bytes(descriptor->xdr_decode_ptr, (char **) &ptr, &xdr_size, size);
33
34    free(key);
35    return (int) TRUE;
36  }
```

7.3.3 Publication et découverte de ressources

Objectif. L'objectif de ce module est de fournir un ensemble de primitives adaptées à la topologie réseau sous-jacente, pour la publication et la découverte de ressources. Le module mémoire haut mais également les couches de tolérance aux fautes requièrent un tel module. Par exemple, lorsqu'un client souhaite accéder à une donnée déjà existante dans JUXMEM[3], le module publication et découverte est utilisé. Une des primitives de ce module va permettre de découvrir l'annonce d'un bloc de données recherché. Cette annonce contient les informations nécessaires en vue de la projection de la donnée ainsi découverte dans l'espace d'adressage du client. La couche de tolérance aux fautes utilise également le module publication et découverte. Par exemple, que des entités d'un réseau JUXMEM soient volatiles et qu'un fournisseur membre du groupe DATA D disparaisse. La couche de tolérance aux fautes va alors rechercher de nouveaux fournisseurs susceptibles de se joindre au groupe DATA D, via ce module.

Interface externe et exemple d'utilisation. L'objet Peer_search de la structure JuxMem_peer implémente l'interface de programmation du module publication et découverte. Elle est décrite ci-dessous.

juxmem_search_cache_lookup. Cette primitive permet de rechercher des annonces dans le cache local du pair. Elle retourne une liste d'annonces répondant aux critères de la requête.

juxmem_search_publish. Cette primitive se décline en trois versions. La première version, suffixée par cache, permet de publier des annonces dans le cache local du pair. La seconde, suffixée par local, permet de publier une annonce à distance sur le gestionnaire dont dépend le pair (s'il n'est pas lui-même gestionnaire). Cette publication s'effectue dans le groupe CLUSTER. La troisième et dernière, suffixée par remote, est similaire à la précédente. Toutefois, la publication a lieu dans le groupe JUXMEM. Toutes les versions de cette primitive prennent en entrée une annonce.

juxmem_search_search. Cette primitive se décline en deux versions. La première, suffixée par local, permet de rechercher une annonce dans le groupe CLUSTER auquel est connecté le pair courant. Elle retourne une liste d'annonces répondant aux critères de la requête. La seconde, suffixée par remote est similaire à la précédente. Toutefois, la recherche a lieu dans le groupe CLUSTER.

Le listing 7.4 présente le code simplifié d'utilisation de ces primitives pour la recherche d'annonces mémoire de la donnée à accéder (extrait de la routine juxmem_mmap). Dans cet exemple, une recherche sur l'identifiant de la donnée et dans le groupe CLUSTER est

[3] Via la primitive juxmem_mmap.

tout d'abord lancée (ligne 8). Puis, le processus client va se bloquer pendant au maximum 5 secondes, en attente d'une réponse (ligne 9). En cas d'absence de réponse (ligne 11), une recherche dans le groupe JUXMEM est alors lancée (ligne 12).

Listing 7.4 – Exemple d'utilisation des primitives du module publication et découverte pour l'implémentation de la routine juxmem_mmap.

```
1  void* juxmem_mmap(JuxMem_peer *peer, void *ptr, size_t len,
2                    const char *data_id, off_t offset)
3  {
4    Jxta_listener *listener = jxta_listener_new(NULL, NULL, 1, 1);
5    Jxta_vector   *vector = NULL;
6    ...
7    jxta_listener_start(listener);
8    juxmem_search_local_search(peer, DISC_ADV, "Id", data_id, 1, listener);
9    jxta_listener_wait_for_event(listener, APR_USEC_PER_SEC * 5,
10                                (Jxta_object **) &vector);
11   if (jxta_vector_size(vector) == 0) {
12     juxmem_search_remote_search(peer, DISC_ADV, "Id", data_id, 1, listener);
13     jxta_listener_wait_for_event(listener, APR_USEC_PER_SEC * 60,
14                                  (Jxta_object **) &vector);
15   }
16   jxta_listener_stop(listener);
17   JXTA_OBJECT_RELEASE(listener);
18   ...
19 }
```

Implémentation. Ce module repose sur le service de découverte de JXTA. Toutefois, les primitives proposées permettent de s'abstraire du groupe JXTA utilisé pour la recherche d'annonces. En effet, elles fournissent la notion de publication/recherche en *local*, c'est-à-dire dans le groupe CLUSTER, et à *distance*, c'est-à-dire dans le groupe JUXMEM. L'algorithme utilisé par JXTA pour la découverte d'annonces est décrit à la section 6.2.2.

7.4 Gestion des communications

Objectif. L'objectif du module communication de *juk* est de fournir des primitives pour l'envoi et la réception de messages entre pairs. Comme nous l'avons décrit à la section 5.2.4.2, notre approche se fonde sur l'utilisation d'identifiants logiques en guise d'adresse des pairs. L'objectif est de s'abstraire de la localisation physique des pairs dans le réseau JUXMEM. Par ailleurs, ce module a également pour objectif de fournir aux couches supérieures de JUXMEM une interface de programmation indépendante de la bibliothèque P2P utilisée. L'ensemble des primitives définies par ce module est notamment utilisé par les protocoles de cohérence et de tolérance aux fautes pour l'envoi de leurs messages sur un réseau JUXMEM.

Interface externe et exemple d'utilisation. Par l'intermédiaire de l'objet Peer_comm de JuxMem_peer, le module communication implémente l'interface de programmation décrite ci-dessous.

juxmem_send(Jxta_id *id, char *tag, Jxta_message *msg).

Cette primitive permet l'envoi du message msg à un identifiant id et sur l'étiquette tag.

Le pair ou l'ensemble de pairs qui écoutent sur cet identifiant doivent avoir enregistré au préalable une fonction de rappel sur l'étiquette tag.

juxmem_multicast_send(Jxta_vector *ids, char *tag, Jxta_message *msg). Cette primitive est similaire à la précédente, sauf que l'envoi est effectué sur un ensemble d'identifiants contenus dans la liste ids.

juxmem_add_tag_listener(char *tag, Jxta_listener *listener). Cette primitive permet d'ajouter localement une fonction de rappel listener qui sera appelée à chaque réception par le pair d'un message sur l'étiquette tag.

juxmem_remove_tag_listener(char *tag). Cette primitive permet de retirer localement la fonction de rappel appelée lors de la réception par le pair de messages sur l'étiquette tag.

juxmem_get_comm_id. Cette primitive permet de connaître l'identifiant sur lequel le pair local écoute.

Les fonctions de rappels associée à la réception de messages permettent de récupérer un pointeur sur un objet de type Peer_comm_event. Trois primitives permettent à partir de cette structure opaque de récupérer : l'identifiant utilisé pour l'émission du message, l'étiquette sur laquelle le message a été reçu ainsi que le message proprement dit.

Le listing 7.5 montre un exemple d'utilisation des primitives du module communication. Le code présenté concerne la réception des messages au niveau de la couche de tolérance aux fautes. Dans cet exemple, une fonction de rappel nommée basic_sog_listener est enregistrée (lignes 1 à 4). Dans l'implémentation de cette fonction (commençant à la ligne 7), l'identifiant du pair qui a émis le message ainsi que le message sont passés en paramètre à la routine de traitement des messages reçus (ligne 12).

Listing 7.5 – Exemple d'utilisation des primitives du module communication pour la réception des messages au niveau de la couche de tolérance aux fautes.

```
1  self->impl->listener = jxta_listener_new((Jxta_listener_func) basic_sog_listener,
2                                           self, 1, -1);
3  jxta_listener_start(self->impl->listener);
4  juxmem_add_tag_listener(peer, self->impl->listener_tag, self->impl->listener);
5  ...
6
7  static void basic_sog_listener(Jxta_object *obj, void *arg) {
8    Peer_comm_event *comm_event = (Peer_comm_event *) obj;
9    sog_client_stub *self = (sog_client_stub *) arg;
10
11   if (self->impl->handler != NULL) {
12     recv_grp_comm(self->impl->handler,
13                   juxmem_comm_event_get_sender_id(comm_event),
14                   juxmem_comm_event_get_message(comm_event));
15   }
16 }
```

Implémentation. Ce module repose actuellement sur le service de canal virtuel des spécifications JXTA. Les propriétés de ce service de communication notamment en termes de fiabilité sont présentées à la section 10.1.3. Chaque pair JUXMEM dispose d'un canal de communication sur lequel il écoute. Ce canal est instancié dans le groupe JUXMEM. Ce canal de

communication est utilisé par l'ensemble des modules de JUXMEM par ajout/retrait d'étiquettes et de fonctions de rappel associées.

Le module communication utilise une table des fonctions permettant de modifier ou de changer son implémentation. L'objectif est de porter facilement cette interface sur le service de communication point-à-point de JXTA par exemple ou en utilisant des canaux bidirectionnels (voir section 10.1).

7.5 Micro-évaluation expérimentale

Dans cette section, nous présentons une micro-évaluation expérimentale et préliminaire de JUXMEM.

Conditions expérimentales. L'environnement matériel de test est la grille expérimentale Grid'5000 [51], avec un maximum de trois sites participant à cette micro-évaluation (Orsay, Lyon et Rennes). Dans les trois sites, les machines utilisées sont équipées de biprocesseurs Opteron d'AMD cadencés à 2,0 GHz, munis de 2 Go de mémoire vive et exécutant la version 2.6 du noyau Linux. Au sein des sites, le réseau de communication utilisé est Gigabit Ethernet. La latence de communication est de 3,5 ms entre les sites d'Orsay et de Lyon, de 4,5 ms entre les sites d'Orsay et de Rennes et de 6,25 ms entre les sites de Lyon et de Rennes. L'ensemble des programmes de micro-évaluations de JUXMEM-C 0.3 a été compilé avec gcc 4.0 (niveau d'optimisation -O2).

Évaluation des performances des opérations de lecture et d'écriture. Le réseau JUXMEM utilisé pour la réalisation de ces mesures est constitué d'un gestionnaire, d'au maximum deux fournisseurs et d'un client C. L'ensemble de ces entités est réparti sur le site de Rennes.

Le client C alloue une zone mémoire A dont la valeur initiale est stockée sur un nombre variable de fournisseurs (noté f et égal à 1 ou 2). Puis cette donnée est projetée dans une autre zone mémoire de son espace d'adressage, notée B (avec $A \neq B$). Le client C alterne alors des écritures sur la totalité de la donnée stockée dans la zone mémoire A, avec des lectures depuis la zone mémoire B. En conséquence, la zone mémoire B doit être intégralement mise à jour à chaque itération du fait de la modification de la donnée dans la zone mémoire A. Lors de ce test, nous mesurons les performances des lectures (c'est-à-dire juxmem_acquire_read, juxmem_release) et des écritures (c'est-à-dire juxmem_acquire, memset, juxmem_release). Par ailleurs, nous mesurons également les performances des écritures lorsque celles-ci sont alternées dans les zones mémoires A et B. À chaque itération, l'intégralité de ces zones mémoires doivent donc être mises à jour. Dans chacune des expériences, nous faisons varier la taille d de la donnée de 4 octets à 8 Mo. L'ensemble des mesures présentées sont calculées sur la base de la moyenne de 100 itérations. Enfin, ces tests sont réalisés avec JUXMEM configuré pour ne pas utiliser l'encodage XDR.

La figure 7.7 présente les résultats obtenus en termes de débit pour les performances des opérations de lecture et d'écriture, pour $f = 1$ et $f = 2$.

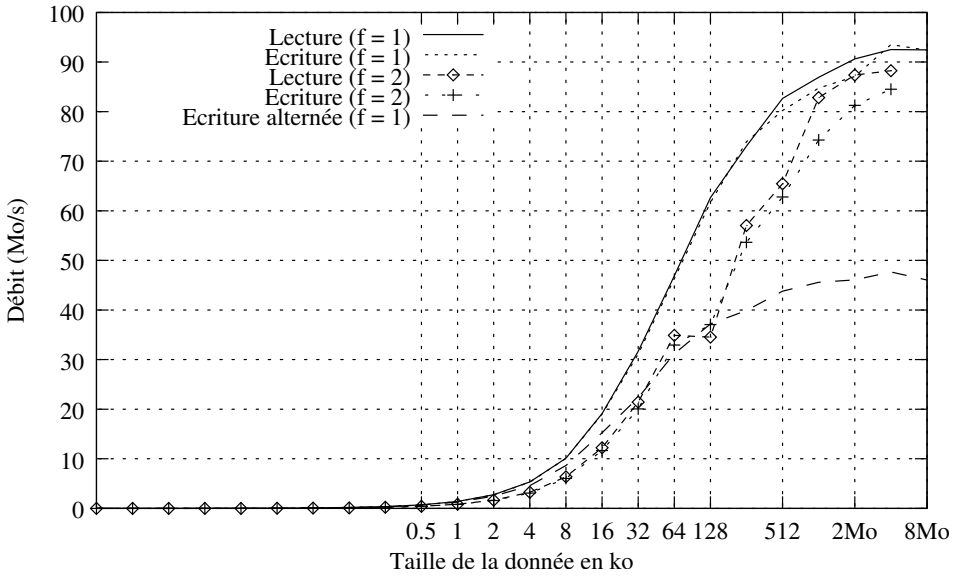

FIG. 7.7 – Débit des opérations de lecture et d'écriture de JUXMEM-C 0.3 pour un nombre variable de fournisseurs (f).

Coût des accès à une donnée non répliquée. Tout d'abord, nous analysons ces résultats lorsque f est égal à 1. Dans ce cas, le débit maximal atteint pour une lecture ou une écriture est de 92 Mo/s, pour des valeurs de d supérieures ou égales à 4 Mo. Par rapport au débit maximal possible de ce type de réseau[4], une opération de lecture ou d'écriture exploite actuellement environ 79 % des capacités. Ainsi, cette version expérimentale de JUXMEM est en mesure d'exploiter les capacités des réseaux de type Gigabit Ethernet pour la gestion des données. Les mauvaises performances pour des petites tailles de données (débit inférieur à 20 Mo/s lorsque d est égal à 32 ko) sont une conséquence du coût important en termes de latence des opérations de lecture et d'écriture. En effet, la latence d'une lecture est de 655 μs et la latence d'une écriture est de 665 μs (voir tableau 7.1). Ces fortes valeurs s'expliquent par : 1) la taille des messages réseaux incluant une donnée (par défaut d'au moins 2048 ko, valeur paramétrable) et 2) le nombre de messages réseau nécessaire pour effectuer une lecture ou une écriture. Dans les deux cas, ce nombre est de quatre.

1. Demande du verrou par le client au fournisseur. La taille totale de ce type de message est de 408 octets.
2. Acquittement de la prise du verrou par le fournisseur au client. Lorsqu'il s'agit de la prise d'un verrou en lecture, et compte tenu de l'alternance entre les écritures et les lectures le fournisseur transmet au client une copie à jour de la donnée. La taille de ce type de message est alors au minimum de 2463 octets pour une taille de donnée applicative de 1 octet (résultat valable jusqu'à une donné applicative d'une taille de 2048 octets). Lorsqu'il s'agit de la prise d'un verrou en écriture et puisque la donnée n'a pas été modifiée, la taille de ce type de message est de 329 octets.
3. Demande de relâchement du verrou par le client au fournisseur. Lorsqu'il s'agit du relâchement d'un verrou en écriture, le client transmets au fournisseur ses modifications.

[4]Performance d'une *socket* C soit 116,5 Mo/s (voir section 10.2.2).

La taille de ce type de message est alors au minimum de 2542 octets pour une taille de donnée applicative de 1 octet (résultat valable jusqu'à une donnée applicative d'une taille de 2048 octets). Lorsqu'il s'agit du relâchement d'un verrou en lecture, la taille de ce type de message est de 408 octets.

4. Acquittement du relâchement du verrou par le fournisseur au client. La taille totale de ce type de message est de 329 octets.

Compte tenu des résultats en termes de latence de la couche de communication de JUXMEM-C (latence de 127 μs pour une taille de message de 1 octet applicatif, voir section 10.2.2), les traitements nécessaires du côté fournisseur représentent au minimum environ 150 μs. Ce coût est expliqué par une implémentation encore expérimentale, et donc encore améliorable, du rôle de fournisseur dans JUXMEM-C 0.3. Notons que lorsque le protocole de cohérence utilisé pour la gestion de la donnée sur le fournisseur est hiérarchique, les performances sont similaires. Un protocole de cohérence hiérarchique vise à favoriser le traitement des requêtes intrasites et d'éviter de coûteuses communications intersites notamment en termes de latence, afin d'augmenter les performances d'accès aux données. Toutefois, nous pouvons constater qu'un tel mécanisme n'introduit pas dans notre test de surcoût. Pour plus d'explications sur le fonctionnement interne de ce protocole de cohérence hiérarchique, le lecteur peut se référer à [127].

Coût des accès à une donnée répliquée. Analysons maintenant le cas où f est égal à deux, c'est-à-dire ou la donnée est répliquée sur deux fournisseurs. Dans notre cas, le client communique avec un des fournisseurs appelé *séquenceur*. Le débit maximal atteint pour une lecture est de 88 Mo/s pour des valeurs de d supérieures ou égales à 4 Mo. Le débit maximal atteint pour une écriture est de 84 Mo/s pour des valeurs de d supérieures ou égales à 4 Mo. La diminution de performance en termes de débit est minime (de 4 à 8 Mo/s). Ce résultat s'explique principalement par la mise à jour du fournisseur non séquencer de manière asynchrone par le fournisseur séquencer. Dans le cas où $f = 2$, la latence est de 1160 μs pour une lecture et de 1100 μs pour une écriture. Ce surcoût important en termes de latence (environ 550 μs) est expliqué par l'utilisation de mécanismes de gestion de la tolérance aux fautes nécessaires du côté fournisseur. Ces mécanismes autorisent la gestion de manière cohérente des différentes copies d'une donnée, afin de garantir la persistance de celle-ci en présence de fautes. Pour plus d'explications sur le fonctionnement interne de ces mécanismes de tolérance aux fautes ainsi que pour des évaluations avec des stratégies de réplications plus complexes, le lecteur peut se référer à [127].

Coût des accès à une donnée non-répliquée lors d'écritures alternées. Par ailleurs, nous avons également mesuré les performances de l'opération d'écriture lorsque des écritures dans les zones mémoires A et B sont alternées (avec $f = 1$). Dans ce cas, les deux zones mémoires du client C doivent être intégralement mises à jour à chaque itération. Ce test permet donc d'émuler la présence d'écrivains multiples sur une même donnée. Le débit maximal atteint est de 47 Mo/s pour d égal à 4 Mo (voir figure 7.7) et latence est de 700 μs. Par rapport aux résultats précédents, ce résultat est conforme à nos attentes compte tenu de la nécessité, à chaque itération de : 1) mettre entièrement à jour les zones mémoires A et B puis 2) de propager les modifications réalisées (totalité de la donnée). Ce résultat suggère

Implémentation	JUXMEM-C 0.3 sans XDR		JUXMEM-C 0.3 avec XDR	
Métrique	Débit	Latence	Débit	Latence
Lecture	92 Mo/s (4 Mo)	655 μs	22 Mo/s (1 Mo)	675 μs
Écriture	92 Mo/s (4 Mo)	665 μs	20 Mo/s (1 Mo)	685 μs

TAB. 7.1 – Performances en termes de débit et de latence de JUXMEM-C 0.3 pour les opérations de lecture et d'écriture. Pour les débits, les chiffres entre parenthèses correspondent à la taille de la donnée pour laquelle le débit maximal est atteint.

Primitive	juxmem_malloc	juxmem_mmap
Coût	3,3 ms	2,5 ms

TAB. 7.2 – Coût des primitives juxmem_malloc et juxmem_mmap au sein d'un site.

qu'un scénario avec de multiples écrivains ne permet d'utiliser que 40 % des capacités d'un réseau de type Gigabit Ethernet.

Coût de l'utilisation de l'encodage XDR. Le tableau 7.1 compare les débits des opérations de lecture et d'écriture lorsque JUXMEM-C est configuré avec ou sans l'utilisation de l'encodage XDR pour la gestion des données. Les débit maximal atteint pour une lecture lorsque l'encodage XDR est utilisé est de 22 Mo/s pour d égal à 1 Mo. Pour une opération d'écriture, le débit maximal atteint est de 20 Mo/s pour une valeur de d égale à 1 Mo. Le coût de l'encodage XDR est donc significatif et pénalise de manière importante les performances en termes de débit d'une opération de lecture et d'écriture. En revanche l'influence de l'utilisation de l'encodage XDR sur la latence est négligeable : celle-ci est autour de 680 μs.

Coût des primitives juxmem_malloc et juxmem_mmap. Nous avons également mesuré le coût des primitives juxmem_malloc et juxmem_mmap. Pour la première primitive, le test consiste pour un client à allouer des données de tailles variables (de 1 octet à 8 Mo) sur un fournisseur s'exécutant dans le même site que le client. Dans ce cas, le coût de cette opération est de 3,3 ms. Pour la seconde primitive, le test consiste pour une client à projeter des données de tailles variables (de 1 octet à 8 Mo) dans son espace d'adressage. Les données ne sont pas répliquées et sont stockées sur un fournisseur s'exécutant dans le même site que le client. Dans ce cas, le coût de la primitive juxmem_mmap est de 2,5 ms. Notons que dans les deux cas, les résultats obtenus ne dépendent pas de la taille de la donnée. Le tableau 7.2 résume ces résultats.

Évaluation des performances de la découverte de fournisseurs. Le réseau JUXMEM utilisé est constitué d'un client, d'un nombre variable f de fournisseurs et d'un nombre variable g de gestionnaires. Chaque entité JUXMEM est exécutée sur une machine différente. Le code du client consiste à mesurer le temps nécessaire t pour découvrir un ensemble r de fournisseurs offrant chacun un espace mémoire d'une taille d, lorsque f et g varient. Notons donc que ce test ne mesure que la première partie du coût de la primitive juxmem_malloc : la découverte d'annonces fournisseurs (voir section 7.3.1 pour plus d'explications). Chaque fournisseur

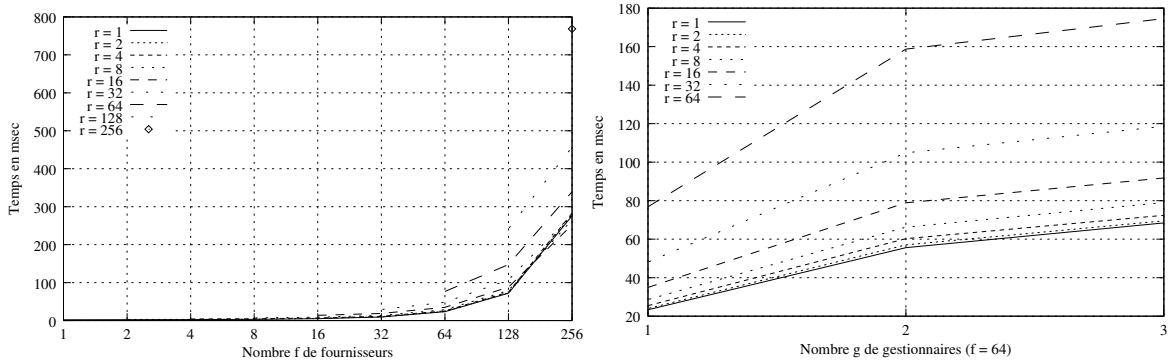

FIG. 7.8 – Temps nécessaire t pour découvrir un nombre variable de fournisseurs r sur un réseau JUXMEM constitué d'un nombre variable de gestionnaires (g) et de fournisseurs (f).

offre un espace mémoire d'une taille supérieure à d. Chaque gestionnaire est exécuté dans un site différent : trois sites de la grille expérimentale sont impliqués dans ce test (Lyon, Orsay et Rennes). L'objectif de cette évaluation est de caractériser la cardinalité maximale d'un groupe CLUSTER préservant les performances de recherche d'espaces mémoire.

La partie gauche de la figure 7.8 présente ce temps nécessaire t lorsque f varie de 1 à 256 (c'est-à-dire que l'ensemble des fournisseurs est connecté à un gestionnaire) et avec $g = 1$. La recherche d'un fournisseur sur un réseau JUXMEM comprenant un fournisseur est de 2 ms. Ce temps s'accroît en fonction du nombre de fournisseurs présents dans le réseau JUXMEM. Pour $f = 256$ et $r = 1$, t est égal à 275 ms. Lorsque $r = 256$, t vaut alors 768 ms (toujours pour $f = 256$). Cet accroissement important de t s'explique par le coût au niveau JXTA des requêtes de recherche d'annonces fournisseurs adéquates sur le cache local du gestionnaire (voir section 7.3.1). Dans nos mesures (c'est-à-dire pour des valeurs de f égales à 64 et 128 et avec r variant de 1 à f par puissance de 2), ce coût au niveau JXTA représente en moyenne 97 % du coût total h de la fonction de traitement des requêtes d'allocation sur les gestionnaires. Pour $f = 64$, $g = 1$ et $r = 64$, h est égal à 27 ms et t est égal à 23,5 ms.

Toutefois, lorsque f est inférieur à 128 la valeur de t est inférieure à 75 ms (pour toute valeur de r inférieure à 4). La valeur de t devient inférieure à 25 ms lorsque f est inférieur à 64 (également pour toute valeur de r inférieure à 4). Comme expliqué dans les évaluations de [127], une valeur de r de 3 est généralement suffisante pour assurer le stockage persistant des données, compte tenu de notre contrainte de volatilité en heures. En conséquence, les résultats de cette expérience montrent que la cardinalité d'un groupe CLUSTER ne doit pas dépasser une valeur de 128, sous peine d'importantes dégradations de performances. Compte tenu des tailles actuelles de certaines grappes de machines ou du nombre de grappes de machines présentes par sites, il est donc nécessaire de configurer JUXMEM pour utiliser plusieurs groupes CLUSTER par site.

La partie droite de la figure 7.8 présente le temps nécessaire t pour découvrir un ensemble r de fournisseurs offrant chacun un espace mémoire d'une taille d, lorsque g varie de 1 à 3 gestionnaires (avec $f = 64$). Ainsi, t vaut 78 ms pour rechercher 16 fournisseurs lorsque g est égal à 2. Comparé à un réseau JUXMEM constitué d'un gestionnaire cela représente un coût multiplié par 2,2 ($t = 35$ ms lorsque $g = 1$). Ce surcoût est principalement expliqué par la nécessité d'effectuer deux communications WAN (voir section 5.2.3.1),

lorsque g est égal à 2 et compte tenu de notre choix de placement des gestionnaires sur des sites différents. Toutefois, lorsqu'un troisième gestionnaire est ajouté un surcoût d'un facteur de seulement 1,1 est observé (soit 13 ms) par rapport à un réseau JUXMEM constitué de deux gestionnaires. Cette faible multiplication du coût est expliquée par la parallélisation des recherches locales sur chacun des gestionnaires dans l'algorithme que nous utilisons pour découvrir des espaces mémoire (voir section 5.2.3.1). Ce résultat permet d'envisager la présence d'un nombre important de groupes CLUSTER, confirmant ainsi les conclusions de la précédente évaluation (paragraphe précédent).

7.6 Conclusion

Ce chapitre a présenté nos implémentations, JUXMEM-C et JUXMEM-J2SE, de notre proposition d'architecture JUXMEM au-dessus des spécifications JXTA. Après un bref aperçu de l'organisation générale de l'implémentation en couches, nous avons détaillé le cœur de JUXMEM-C : le noyau *juk*. Nous avons présenté les différents modules qui le constituent et nous avons précisé leurs rôles. L'objectif de *juk* est : 1) d'implémenter l'interface de programmation du concept de service de partage de données et 2) de fournir des mécanismes P2P adaptés aux infrastructures des grilles de calcul et utilisables par les couches supérieures de JUXMEM de gestion de la cohérence et de la tolérance aux fautes. Enfin, nous avons présenté une micro-évaluation expérimentale des performances de JUXMEM et du coût de certaines opérations de *juk*. Nous avons notamment donné une évaluation préliminaire des performances en termes de débit et de latence des opérations de lecture et d'écriture de JUXMEM. Cette évaluation a permis de montrer que JUXMEM est en mesure d'exploiter jusqu'à 79 % des capacités d'un réseau Gigabit Ethernet, en atteignant des débits de l'ordre de 90 Mo/s pour les opérations d'écriture et de lecture. Des évaluations plus poussées, notamment à plus grande échelle, avec mesure des débits agrégés des opération de lecture et d'écriture par exemple, sont nécessaires pour confirmer ces résultats initiaux satisfaisants. Par manque de temps, nous n'avons pas pu les réaliser, mais ce travail est en cours. Par ailleurs, cette micro-évaluation a également permis d'évaluer la cardinalité adéquate des groupes CLUSTER garantissant que les performances de l'algorithme de recherche des fournisseurs ne soient pas dégradées. Enfin, notons que l'évaluation que nous avons réalisée est basée sur une version de JUXMEM-C (0.3) dont l'implémentation du rôle de fournisseur est préliminaire et donc améliorable.

Les *module mémoire* de *juk* permet une gestion de l'espace mémoire à la fois sur les fournisseurs mais également sur les clients. Cette gestion est réalisée de manière transparente pour les protocoles de cohérence, grâce à l'utilisation de primitives bien spécifiées. Par ailleurs, *juk* met également à disposition un *module de communication entre pairs* qui offre une gestion de communications dynamique, c'est-à-dire supportant des entitées volatiles grâce à l'infrastructure P2P sous-jacente utilisée. Enfin, un *module de publication et de découverte* offre des primitives pour la découverte d'espaces mémoire, nécessaires pour le stockage de données mais également la mise en œuvre des mécanismes de réplication des données. La version la plus aboutie de *juk* représente 7 500 lignes de code dans le langage C.

Le logiciel JUXMEM (JUXMEM-C et JUXMEM-J2SE) a été rendu disponible sous licence GNU LGPL et déposé à l'Agence pour la protection des Programmes (APP)[5]. JUXMEM dans

[5]Sous le numéro IDDN.FR.001.490010.000.S.P.2004.000.10600.

sa dernière version (0.3) est disponible au téléchargement [218]. L'intégration et la validation de JUXMEM dans deux paradigmes de programmation utilisés pour le développement d'applications scientifiques pour grilles de calcul font l'objet de la partie suivante de ce manuscrit.

Troisième partie

Validation du service JUXMEM

Chapitre 8

Utilisation et évaluation de JUXMEM dans le modèle Grid-RPC

Sommaire

Dans cette troisième partie du manuscrit, nous présentons l'utilisation et l'évaluation de JUXMEM au sein de deux modèles de programmation : le modèle Grid-RPC (ce chapitre) et les modèles composant (chapitre 9).

Le premier chapitre de cette partie décrit l'utilisation de JUXMEM dans le modèle de programmation Grid-RPC. Dans un premier temps, le modèle Grid-RPC et les propositions actuelles pour la gestion des données dans ce modèle sont présentés. Au vu des lacunes des différentes solutions, nous proposons dans un deuxième temps l'utilisation de notre concept de service de partage de données pour la gestion des données persistantes dans ce modèle de programmation. Nous détaillons l'utilisation de JUXMEM au sein de l'intergiciel DIET, qui implémente le modèle Grid-RPC. Enfin, dans un dernier temps, une évaluation des bénéfices de notre proposition pour la gestion des données est présentée. Cette évaluation est effectuée via un code synthétique. Par ailleurs, nous avons également exécuté avec succès

une application réelle s'exécutant au-dessus de l'intergiciel Grid-RPC DIET et utilisant JUXMEM pour la gestion de ses données. Ce travail a été réalisé en collaboration avec Eddy Caron[1].

8.1 Le modèle de programmation Grid-RPC

Le modèle Grid-RPC [155] est né d'un effort du *Global GridForum* (GGF) pour standardiser et implémenter, dans le contexte des grilles de calcul, le modèle de programmation par appel de procédures (*Remote Procedure Call*, RPC [131]). Comparé au modèle RPC classique d'Unix, le modèle Grid-RPC inclut également la possibilité d'effectuer des tâches parallèles à gros grain de manière asynchrone. Les requêtes de calcul à distance peuvent en effet être traitées par des serveurs séquentiels ou parallèles. Toutefois, ce parallélisme potentiel au niveau des serveurs n'est pas visible au niveau des clients.

Le modèle Grid-RPC a été défini dans le cadre du groupe de travail *GRIDRPC-WG* [212], dirigé par Craig Lee. L'objectif visé par ce groupe de travail est de définir la syntaxe ainsi que la sémantique des opérations disponibles dans l'interface de programmation du côté client de ce modèle [155]. Ainsi, les programmeurs d'applications basées sur le modèle Grid-RPC peuvent utiliser différentes implémentations sans avoir à se soucier de la portabilité de leur code. Cette spécification précise également les entités impliquées dans le modèle Grid-RPC ainsi que leur rôle. Le modèle Grid-RPC vise à mettre en place une infrastructure logicielle pour l'utilisation des ressources de calcul d'une grille par différentes applications s'exécutant de manière concurrente. Nous appelons *plate-forme de calcul* (en anglais *Network Enabled Server*, NES) un intergiciel déployé sur une grille de calcul qui offre l'interface de programmation du modèle Grid-RPC aux développeurs d'applications. Des exemples de tels intergiciels sont par exemple DIET [54], Ninf [130], NetSolve [26], OmniRPC [154] ou XtremWeb [52]. Dans la suite de ce manuscrit nous les notons *intergiciel Grid-RPC*.

La figure 8.1 présente de manière conceptuelle les différentes entités du modèle Grid-RPC ainsi que leurs interactions sous la forme d'étapes. Un *service* s'enregistre dans un *répertoire* afin d'être disponible dans la plate-forme de calcul (étape 1). Notons qu'un service est hébergé au sein d'un processus d'une machine appelée serveur et que plusieurs services peuvent s'exécuter dans un processus (par utilisation de processus légers). Par ailleurs, un service peut par exemple lancer un ensemble de processus MPI afin de résoudre un problème. Pour simplifier cette multitude de configurations possibles et par abus de langage, nous utilisons le terme *serveur* pour désigner le processus d'un serveur qui héberge potentiellement plusieurs services. Afin de découvrir le service qu'il recherche, un processus *client* d'une machine contacte un répertoire (étape 2) qui va éventuellement lui retourner une *référence de fonction* sur le service recherché (étape 3) s'il est connu. Notons que le modèle Grid-RPC ne spécifie pas les mécanismes à utiliser pour la découverte des services. Le client peut alors utiliser cette référence de fonction pour appeler le service (étape 4) qui va effectuer le calcul et éventuellement retourner un résultat (étape 5). À l'étape 3, l'objectif des intergiciels Grid-RPC est de trouver un serveur approprié (si ce n'est le meilleur !) pour chacune des requêtes à traiter, et cela parmi un ensemble de serveurs candidats. Ce choix s'effectue en prenant en compte des informations sur l'efficacité des différents serveurs disposant du service recherché. Ces informations peuvent être statiques, telles que par exemple la fréquence

[1] Maître de conférences à l'ENS Lyon au sein du projet GRAAL.

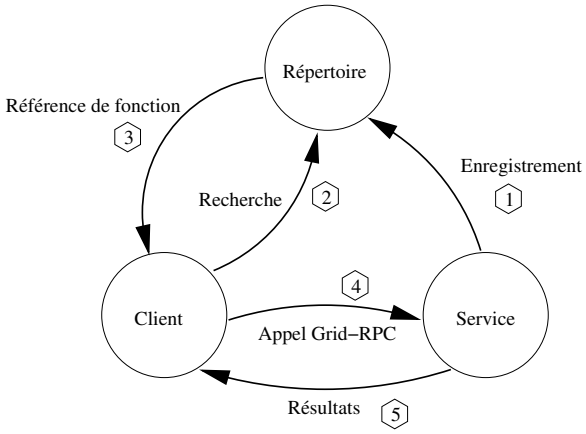

FIG. 8.1 – Illustration des entités conceptuellement présentes dans le modèle Grid-RPC.

du processeur de la machine et la taille de sa mémoire vive, mais également dynamiques comme les services installés, la charge de la machine, la localisation des données nécessaires aux calculs, etc. Cet équilibrage de charge au sein de la plate-forme de calcul est réalisé par un ou des *agents* qui implémentent de manière distribuée la fonctionnalité de répertoire. L'objectif de ces agents est alors d'optimiser le débit global de la plate-forme de calcul.

Dans la terminologie Grid-RPC, une référence de fonction (en anglais *function handle*) représente une association entre une chaîne de caractères correspondant au nom d'un service, appelé également problème, et une instance d'une fonction implémentant ce service sur un serveur. Une fois l'association entre une fonction et un serveur établie, tous les appels à cette fonction sont réalisés sur le serveur ainsi spécifié. Dans la suite de ce chapitre, nous utilisons les termes *appel Grid-RPC* pour désigner un appel sur une telle référence de fonction distante. Un identifiant de session est associé à chaque requête non bloquante et permet de récupérer l'état de la requête, d'attendre la fin de l'appel, d'annuler la requête, d'obtenir le code d'erreur de l'appel, etc. À partir de ces deux concepts, l'interface de programmation du modèle Grid-RPC est principalement constituée des deux fonctions suivantes : grpc_call et grpc_call_async qui permettent respectivement d'effectuer un appel Grid-RPC d'une manière synchrone et asynchrone. Le lecteur intéressé par une description plus détaillée de cette interface de programmation peut se référer à [131].

Par ailleurs, la plupart des intergiciels Grid-RPC (DIET et Ninf-G au moins) spécifient trois *modes d'accès* pour les paramètres d'un appel Grid-RPC. Les paramètres en entrée d'un calcul sont notés être des données en in. Ils ne sont pas modifiables. Les paramètres produits par un calcul sont indiqués être des données en out. Enfin, les paramètres qui peuvent être modifiés pendant un calcul sont des données en inout. Toutefois, le concept de mode d'accès d'un paramètre ne fait pas partie de la spécification du modèle Grid-RPC.

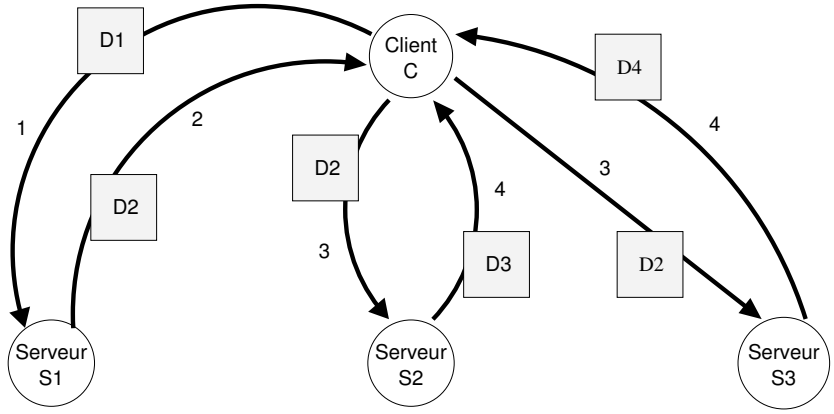

FIG. 8.2 – Illustration de la gestion des données dans l'état actuel de la spécification du modèle Grid-RPC pour une série de requête de calculs.

8.2 Gestion des données dans le modèle de programmation Grid-RPC

Dans cette section, nous détaillons les propositions actuelles pour la gestion des données dans ce modèle ainsi que leurs limites.

8.2.1 Besoins pour la gestion de données

La première version de la spécification du modèle Grid-RPC s'est principalement concentrée sur la gestion des calculs, oubliant volontairement, dans un premier temps, la gestion des données. Ainsi, les données en out ou inout pour un calcul sont actuellement retournées au client, alors que les données en in sont détruites. Il en résulte que d'inutiles mouvements de données sont réalisés, si celles-ci sont utilisées soit par la suite pour d'autres calculs, soit par plusieurs serveurs simultanément. Pour illustrer ce phénomène, prenons un exemple où un client C utilise successivement deux ensembles de serveurs pour effectuer plusieurs calculs. Le premier ensemble de serveur fait intervenir le serveur $S1$, alors que le deuxième ensemble fait intervenir les serveurs $S2$ et $S3$. La figure 8.2 présente cet exemple. Nous faisons l'hypothèse que les calculs réalisés par le deuxième ensemble de serveurs dépendent du premier : la donnée en out $D2$ produite sur $S1$ est utilisée comme donnée en in pour les calculs ordonnancés sur les serveurs $S2$ et $S3$. Nous faisons également l'hypothèse que $D2$ est une donnée intermédiaire qui n'est pas utile pour le client C. En utilisant le modèle Grid-RPC tel qu'il est actuellement défini, le client C récupère la donnée en out $D2$ produite par le premier calcul sur le serveur $S1$ (étape 2). Puis, pour effectuer les deux appels Grid-RPC en parallèle, il doit la transférer à nouveau à la fois vers le serveur $S2$ mais également vers $S3$ (étapes 3). La donnée $D2$ est donc transmise inutilement trois fois entre le client et la plate-forme de calcul. Lorsque les calculs sont terminés, les données $D3$ et $D4$ produites par les serveur $S2$ et $S3$ sont retournées au client C (étapes 4).

Une liste de besoins a donc été établie par le GGF pour pallier les limites actuelles concernant la gestion des données dans le modèle Grid-RPC. Elle comporte les points suivants :

– le passage par référence des arguments partagés entre plusieurs appels Grid-RPC, ce qui peut éventuellement nécessiter une gestion de la cohérence des données en cas d'accès concurrents ;
– l'optimisation de la bande passante lors des transferts de données, par exemple en copiant les sous-parties des données qui sont uniquement nécessaires pour effectuer les calculs ;
– une tolérance aux fautes de la plate-forme de calcul, et donc un stockage persistant tolérant aux fautes[2].

Ajoutons à cela deux autres propriétés qui nous apparaissent indispensables :
– une localisation transparente des données stockées dans le système par les clients pour le passage à l'échelle des applications s'appuyant sur le modèle Grid-RPC ;
– une compatibilité avec l'interface de programmation du modèle Grid-RPC telle qu'elle existe afin d'éviter des modifications importantes dans les applications déjà bâties sur ce modèle.

8.2.2 Application motivante

Pour illustrer les besoins énumérés à la section précédente, nous avons choisi une application d'expertise destinée à la résolution de systèmes linéaires par des solveurs parallèles. Cette application s'appelle Grid-TLSE [67] et met à disposition de ses utilisateurs différents scénarios d'utilisation. Ces scénarios d'expertise permettent aux utilisateurs d'effectuer de manière concurrente plusieurs exécutions utilisant différents solveurs disponibles dans l'application et/ou différentes combinaisons de paramètres pour chaque solveur. Ce type d'application, dit *multiparamétrique*, tire ainsi parti des ressources de calculs mises à disposition par les grilles, en parallélisant les calculs sur différentes machines. Plus concrètement, un utilisateur peut récupérer des statistiques sur l'exécution de plusieurs algorithmes sophistiqués de matrices creuses sur un ensemble de données en entrée. Ces données peuvent être soit les problèmes soumis par l'utilisateur, soit des exemples représentatifs de matrices choisies parmi une collection de matrices. De manière générale, la taille des matrices varie de quelques mégaoctets à quelques centaines de mégaoctets. L'application Grid-TLSE utilise un intergiciel Grid-RPC pour distribuer sur la grille l'ensemble des calculs ainsi générés, également appelés ensemble de *tâches*. Celles-ci consistent en une exécution d'un algorithme d'un solveur creux parallèle, tels que MUMPS [18], SuperLU [70] ou UMFPACK [66], sur une matrice avec des paramètres fixés pour l'algorithme choisi. L'application Grid-TLSE est financée par le programme de recherche de l'Action Concertée Incitative (ACI) GRID, et est développée au sein du laboratoire IRIT (Institut de Recherche en Informatique de Toulouse).

Un scénario typique de l'application Grid-TLSE consiste en la résolution du problème $Ax = b$. De nombreux algorithmes existent pour cela et leur utilisation dépend de la structure de la matrice A, selon qu'elle est pleine ou creuse. Grid-TLSE considère le cas où A et b ont une structure creuse et utilise alors, pour la résolution, un algorithme direct basé sur des techniques de factorisation. L'algorithme considéré se décompose en trois phases : analyse, factorisation et résolution. Nous considérons dans ce scénario une factorisation particulière $A = LU$. L'objectif des différents algorithmes est alors d'exploiter les propriétés structurelles et numériques de la matrice initiale A et éventuellement de la modifier afin

[2] L'objectif n'est toutefois pas de stocker les données de manière pérenne.

d'améliorer ses propriétés pour une meilleure résolution. De telles modifications concurrentes par plusieurs serveurs doivent s'effectuer d'une manière cohérente. Par ailleurs, les résultats produits par une phase de calculs peuvent être partagés lors de la phase suivante. Les données ainsi partagées doivent être migrées ou répliquées sur différents serveurs et être accédées de manière cohérente en lecture. De manière similaire, lors de la phase d'analyse qui consiste à évaluer l'impact de la permutation choisie pour la résolution du problème, un grand nombre ou toutes les permutations dérivant du problème peuvent être considérées. Différentes métriques entrent alors en jeu pour évaluer la performance de chaque permutation : FLOPS réels, utilisation mémoire, temps de calcul, etc. Ainsi, pour tester n différentes permutations et cela avec m différents solveurs, $m \times n$ exécutions doivent être réalisées. La matrice à permuter doit donc être envoyée $m \times n$ fois vers les serveurs utilisés pour effectuer les calculs. Enfin, le nombre de couples solveurs/permutations étant potentiellement grand, le nombre de machines utilisées pour effectuer les calculs peut également être grand. Cela accroît la probabilité de défaillances d'apparaître dans la plate-forme de calcul, rendant également nécessaire l'utilisation d'algorithmes de tolérance aux fautes pour la gestion des données. Toute la gestion des données que nous venons de décrire est actuellement à la charge du code applicatif.

8.2.3 Propositions existantes pour la gestion des données

Dans cette section, nous détaillons les travaux réalisés pour la gestion des données dans le modèle Grid-RPC, en analysant leur adéquation avec les besoins exprimés dans la section précédente.

La première proposition pour la gestion de données repose sur le concept de *request sequencing* [27]. Cette fonctionnalité consiste en l'ordonnancement d'une séquence d'appels effectués par un client sur un serveur. Une séquence est identifiée par les mots-clés begin_sequence et end_sequence. Les transferts de données dus aux dépendances entre les requêtes de calcul encadrées par ces mot-clés sont alors optimisés. Cette proposition a été implémentée dans les intergiciels Grid-RPC NetSolve et Ninf. Toutefois, la principale limite du *request sequencing* est l'impossibilité de redistribuer les données entre différents serveurs. En conséquence, la résolution d'une séquence en parallèle sur deux serveurs différents ne peut pas être effectuée. Dans [72], les auteurs ont étendu NetSolve pour prendre en compte ce problème et implémenter la redistribution des données entre serveurs. Les algorithmes d'ordonnancement ont également été étendus afin de prendre en compte le placement des données. Néanmoins, cette solution introduit une nouvelle opération dans l'interface de programmation du modèle Grid-RPC : la fonction de redistribution des données. Celle-ci doit être explicitement appelée par le code de l'application basée sur l'intergiciel Grid-RPC NetSolve.

Une autre approche pour la gestion des données repose sur l'utilisation d'une infrastructure de stockage distribuée, telle que *Internet Backplane Protocol* [33] (IBP) qui a été décrite à la section 3.2.2. NetSolve a ainsi été modifié pour illustrer cette proposition. Dans ce cas, le client envoie ses données à des serveurs de stockage, et les serveurs de calculs les récupèrent à la demande, lorsqu'ils en ont besoin. Toutefois et comme nous avons pu le constater au chapitre 3, l'utilisation d'une telle infrastructure de stockage ne permet pas de résoudre les problèmes de cohérence des différentes copies, ni de garantir la tolérance aux fautes. Ces mécanismes restent à la charge du code des clients.

L'intergiciel Grid-RPC DIET peut également utiliser un système appelé *Data Tree Manager* (DTM) [80] pour stocker de manière persistante des données. Chaque serveur de calcul dispose d'une liste des données persistantes qu'il stocke, et chaque agent connaît cette liste pour l'ensemble des serveurs qu'il gère. La localisation des données est alors prise en compte pour effectuer l'ordonnancement des requêtes sur la plate-forme de calcul. Ainsi, lors de l'exécution proprement dite des calculs, les données sont automatiquement transférées si certains serveurs ne disposent pas de copie. Par ailleurs, pour réaliser ce stockage persistant, cette proposition spécifie différents *modes de persistance* pour une donnée :

- *volatile*, rattachée à un serveur ;
- *persistante* (noté PERSISTENT) ;
- ou *persistante mais retournée au client* à la fin de l'appel Grid-RPC (noté PERSISTENT_RETURN).

Les problèmes de tolérance aux fautes ainsi que de cohérence des différentes copies n'ont pas été pris en compte dans la conception et l'implémentation de DTM. De plus, cette solution est une implémentation *ad hoc* dans DIET, ce qui interdit un partage des données entre plusieurs intergiciels Grid-RPC.

Toutefois, suite aux travaux sur DTM, des discussions ont commencé en cours pour standardiser la gestion des données dans le modèle Grid-RPC. Ce travail se fonde sur la notion de référence de données (en anglais *data handle*) qui décrit une donnée, sa localisation ainsi que son mode de persistance. Outre la possibilité de référencer une donnée stockée dans un système externe, une localisation transparente des données est possible. Cependant, un client qui souhaite accéder à une donnée produite par un serveur doit explicitement appeler une fonction de lecture afin d'en disposer une copie dans son espace d'adressage. Ainsi, le client n'est pas entièrement déchargé de la gestion des données. De plus, les problèmes de cohérence des données et de tolérance aux fautes ne sont toujours pas pris en compte.

8.3 Utilisation de JUXMEM au sein de l'intergiciel Grid-RPC DIET

La section précédente a permis de constater que les propositions existantes pour la gestion de données dans le modèle Grid-RPC ne permettent pas répondre à l'ensemble des besoins mentionnés dans la section 8.2.1. Or, notre concept de service de partage de données pour grille permet de fournir l'ensemble de ces propriétés. Ainsi, l'utilisation d'un tel concept pour la gestion de données dans le modèle Grid-RPC est naturelle. L'objectif est alors de pallier les limites des différentes propositions actuelles.

Dans cette section, nous décrivons notre proposition d'utilisation de JUXMEM au sein de l'intergiciel Grid-RPC DIET (pour *Distributed Interactive Engineering Toolbox*). Cette plate-forme de calcul est développée à l'ENS Lyon au sein du projet de recherche GRAAL. La dernière version de DIET (2.1) date du 17 février 2006 et représente environ 40 000 lignes de code dans le langage C++.

8.3.1 Présentation de l'environnement DIET

La figure 8.3 représente l'architecture de DIET, composée de plusieurs entités décrites ci-dessous.

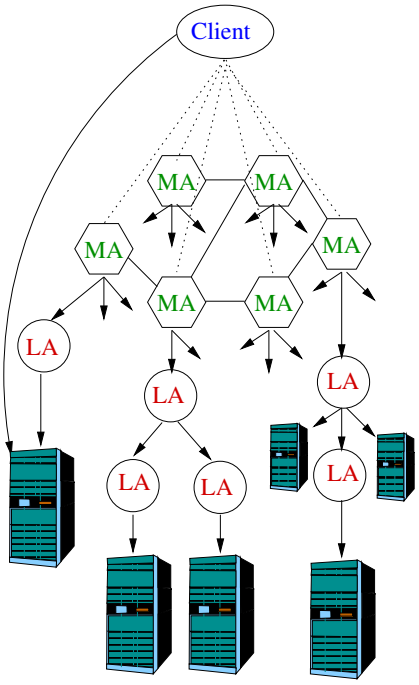

FIG. 8.3 – L'organisation hiérarchique des entités DIET.

Les clients sont des applications qui utilisent DIET afin de résoudre des problèmes. Un problème s'exprime sous la forme d'une requête de calcul qui est une description générique de celui-ci et qui est soumise à la plate-forme de calcul pour traitement. Différents types de clients sont capables d'utiliser DIET, les problèmes pouvant être soumis depuis une page Web, d'un environnement de résolution (en anglais *Problem Solving Environment*, PSE), tel que Scilab [242], ou directement depuis un programme compilé.

L'agent maître (en anglais *Master Agent*, MA) reçoit des requêtes de calcul de la part de différents clients. Le MA a alors pour rôle de propager chacune des requêtes au sein de la plate-forme de calcul et de retourner à chaque client une liste des meilleurs serveurs qui peuvent traiter la requête.

L'agent local (en anglais *Local Agent*, LA) transmet les requêtes de l'agent dont il dépend (LA ou MA) aux serveurs disponibles et aux LA dont il a la charge. Il a également pour rôle de faire « remonter » les informations sur l'état des serveurs vers l'agent dont il dépend. L'originalité de DIET par rapport aux autres intergiciels Grid-RPC se situe dans une gestion décentralisée et hiérarchique de ces agents. Chaque LA stocke la liste des requêtes en cours de traitement, ainsi que le nombre de ses serveurs qui peuvent satisfaire une requête donnée.

Le serveur démon (en anglais *Server Daemon*, SeD) est un processus qui encapsule un ensemble de services et correspond au terme serveur dans notre terminologie (voir section 8.1). Un SeD stocke la liste des services dont il dispose. Il la rend disponible à son LA, et fournit aux éventuels clients une interface pour soumettre leurs requêtes. Enfin, un SeD teste de manière périodique son état : charge instantanée, mémoire disponible, ressources disponibles, etc. En fonction de cet état, le SeD peut fournir à son LA une

prévision de sa performance pour une requête donnée.

Avant de soumettre une requête de calcul, un client DIET doit obtenir une liste de références vers les SeD les plus adéquats. Le client transmet donc sa requête au MA auquel il est connecté. Le MA vérifie tout d'abord si la requête est valide, c'est-à-dire si tous les paramètres sont fournis. Puis, il propage la requête dans l'arborescence des agents jusqu'aux SeD disponibles dans la plate-forme de calcul qui fournissent le service recherché. Chaque SeD retourne à son LA sa performance. À partir de ces informations, chaque LA sélectionne le meilleur SeD et transmet son nom et sa performance à son LA ou MA. Le MA racine agrège les meilleurs SeD trouvés et les ordonne selon leur état et leur disponibilité. La liste ainsi créée est transmise au client qui peut successivement contacter chaque SeD. Dès qu'un SeD répond, le client déplace les données en entrée de la requête (in et/ou inout) sur le SeD sélectionné. Enfin, le SeD exécute la requête du client et retourne les éventuels résultats (donnée en inout et/ou out).

8.3.2 Mise en œuvre de l'utilisation de JUXMEM

L'utilisation de JUXMEM par DIET permet de découpler la gestion des données des calculs ainsi que de bénéficier de manière transparente d'un partage cohérent, un stockage persistant et une tolérance à la volatilité. La mise en œuvre de l'utilisation de JUXMEM par DIET s'appuie sur les modes d'accès qu'il est possible de spécifier pour les données dans DIET. Toutefois, ceci s'applique uniquement à des données qui sont désignées par le programmeur comme persistantes, c'est-à-dire utilisées pour la réalisation de différents calculs. Ainsi, dans la suite de cette section, nous nous intéressons uniquement aux données dont le mode de persistance est à PERSISTENT ou PERSISTENT_RETURN[3].

Avant de présenter l'ensemble des cas de figure possibles (selon les modes d'accès, le type de l'entité DIET impliquée, etc.), prenons un exemple d'utilisation de JUXMEM par DIET. La figure 8.4 représente les entités DIET et JUXMEM utilisées pour la multiplication de deux matrices A et B, constituées de double, sur un SeD $S1$. Ce serveur a été préalablement choisi selon le procédé décrit à la section précédente. La requête pour ce calcul est émise par le client $C1$. Le résultat de ce calcul produit la matrice C. Ainsi, les matrices A et B sont des paramètres en in, et la matrice C est un paramètre en out. Par ailleurs, nous faisons l'hypothèse que les matrices A et B doivent être stockées de manière persistante. Quant à la matrice C, elle est également stockée de manière persistante, mais le client C souhaite en obtenir une copie. Les modes de persistance des matrices A, B et C sont donc respectivement PERSISTENT, PERSISTENT et PERSISTENT_RETURN. Chacune des entités DIET impliquées (client et SeD) charge et utilise la bibliothèque JUXMEM au sein de son processus qui agit ainsi en tant que client vis-à-vis de JUXMEM. Les clients JUXMEM ainsi créés vont alors se connecter à un réseau JUXMEM existant afin de pouvoir stocker et partager des données. En conséquence, à la fois le client et le SeD DIET vont pouvoir utiliser l'interface de programmation offerte par JUXMEM pour allouer de l'espace mémoire pour stocker les matrices, les projeter dans l'espace mémoire du SeD en vue d'effectuer le calcul, acquérir les verrous associés, etc. Notons que les MA et LA n'utilisent pas JUXMEM.

Les listings 8.1 et 8.2 présentent respectivement le code utilisé du coté client et du côté serveur pour l'exemple précédent. Par souci de lisibilité, ce code est simplifié et se focalise

[3] Voir section 8.2.3 pour une description de la différence entre les deux modes.

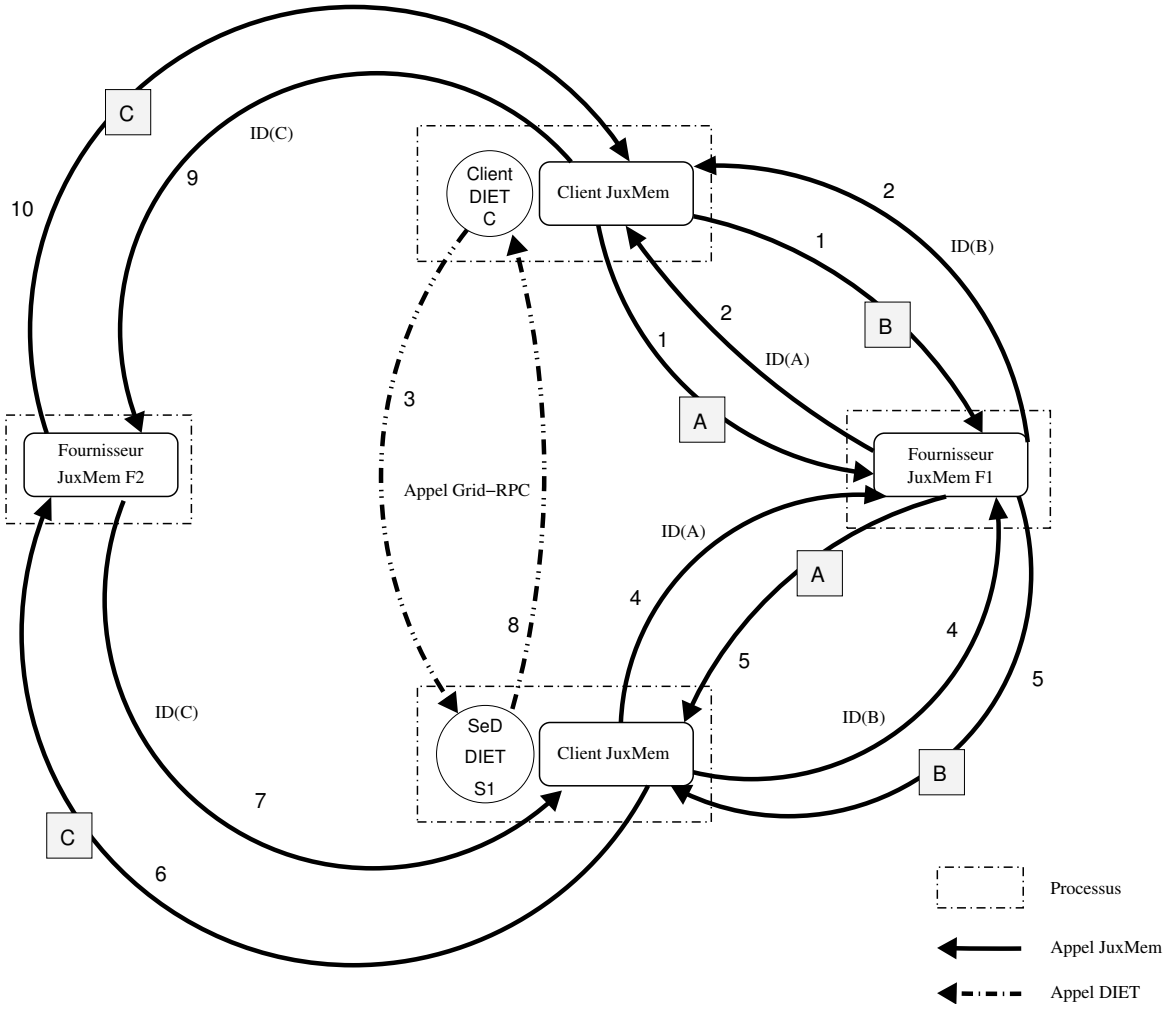

FIG. 8.4 – Multiplication de deux matrices par un client DIET utilisant JUXMEM pour la gestion des données.

sur l'utilisation, par DIET, de l'interface de programmation de JUXMEM, telle qu'elle a été spécifiée au chapitre 5. Notons que le code ici présenté est interne à DIET. Du côté client, il est inclus dans le corps de la fonction grpc_call (et grpc_call_async) et du côté serveur dans corps de la fonction solve qui reçoit les requêtes de calcul à traiter. La gestion des données est donc transparente pour les applications basées sur l'intergiciel Grid-RPC DIET

Listing 8.1 – Code interne au client DIET et lié à JUXMEM pour la multiplication de deux matrices persistantes A et B sur un SeD.

```
1  grpc_error_t grpc_call (grpc_function_handle_t *handle) {
2    grpc_serveur_t *SeD = request_submission(handle);
3    ...
4    char *idA = juxmem_attach(handle->A, data_sizeof(handle->A));
5    char *idB = juxmem_attach(handle->B, data_sizeof(handle->B));
6    ...
7    char *idC = SeD->remote_solve(multiply, idA, idB);
8    ...
9    juxmem_mmap(handle->C, data_sizeof(handle->C), idC);
10   juxmem_acquire_read(handle->C);
11   juxmem_release(handle->C);
12   ...
13 }
```

Listing 8.2 – Code interne au SeD DIET et lié à JUXMEM pour la multiplication de deux matrices persistantes A et B depuis un client.

```
1  char* solve (grpc_function_handle_t *handle, char *idA, char *idB) {
2    double *A = juxmem_mmap(NULL, data_sizeof(handle->A), idA);
3    double *B = juxmem_mmap(NULL, data_sizeof(handle->B), idB);
4
5    juxmem_acquire_read(A);
6    juxmem_acquire_read(B);
7
8    double *C = multiply( A, B);
9
10   juxmem_release(A);
11   juxmem_release(B);
12   return idC = juxmem_attach(C, data_sizeof(handle->C));
13 }
```

Pour revenir à la figure 8.4, du côté client les matrices en in sont tout d'abord projetées dans JUXMEM sur le fournisseur $F1$ via la primitive juxmem_attach (étapes 1 et 2, lignes 4 et 5 du listing 8.1). Les identifiants ainsi créés, $id(A)$ et $id(B)$, sont alors inclus dans la requête de calcul qui est transmise au SeD $S1$ par l'appel Grid-RPC remote_solve (étape 3 et ligne 7 du listing 8.1). Sur ce SeD, les identifiants transmis sont utilisés pour projeter les matrices dans l'espace mémoire local via les appels à la primitive juxmem_mmap (étapes 4 et 5, lignes 2 et 3 dur listing 8.2). Ensuite et en vue du calcul, les verrous associés aux matrices A et B sont pris en lecture via les appels à la primitive juxmem_acquire_read (non représentés sur la figure, lignes 5 et 6 du listing 8.2). La multiplication des deux matrices peut alors être effectuée via l'appel à la fonction multiply ce qui produit la matrice C (ligne 8 du listing 8.2). Les verrous sur les matrices A et B sont alors relâchés via les appels à la primitive juxmem_release (également non représentés sur la figure, lignes 10 et 11 sur le listing 8.2).

	Client		SeD	
	Avant Calcul	Après Calcul	Avant Calcul	Après Calcul
in	attach; msync; detach;		mmap; acquire_read;	release; unmap;
inout	attach; msync;	acquire_read; release;	mmap; acquire;	
out		mmap; acquire_read; release;		attach; msync; unmap;

TAB. 8.1 – Utilisation de l'interface de programmation de JUXMEM par DIET pour des paramètres persistants en in, inout et out, du côté client et SeD, avant et après un calcul. Le préfixe juxmem a été omis.

Puis, la matrice C est projetée dans JUXMEM sur le fournisseur $F2$[4], toujours via la primitive juxmem_attach (étapes 6 et 7, ligne 11 du listing 8.2). L'identifiant ainsi généré est retourné au client C (étape 8), qui peut localement projeter la matrice dans son espace d'adressage (étapes 9 et 10, ligne 9 du listing 8.1). La matrice C ayant un mode de persistance positionné à PERSISTENT_RETURN, la donnée doit être présente dans l'espace d'adressage du client C. Cela est réalisé par l'acquisition puis le relâchement du verrou associé (lignes 10 et 11 du listing 8.1).

La table 8.1 généralise et résume l'utilisation de l'interface de programmation de JUXMEM par DIET pour chaque cas : selon le mode du paramètre, si le calcul a été effectué ou non, et cela à la fois du côté client et du côté SeD. Notons que dans les cas inout et out, après le calcul et du côté client, les appels à JUXMEM ne sont réalisés que si le mode de persistance de la donnée est PERSISTENT_RETURN.

Les modifications apportées à l'intergiciel Grid-RPC DIET pour permettre l'utilisation de JUXMEM pour gérer les données persistantes sont minimes : elles représentent 200 lignes de code dans le langage C++. Les lignes de code en question sont facilement identifiables en raison de #define activés ou non lors de l'installation de DIET. L'implémentation de JUXMEM utilisée est JUXMEM-C, qui est liée aux bibliothèques DIET clientes et SeD que les binaires applicatifs utilisent. Plus précisément, il s'agit de l'interface C++ de JUXMEM-C qui offre la même interface de programmation que sa version C (les appels sont *inlinés*). La première version de DIET qui permet d'utiliser JUXMEM (2.0) date du 21 juillet 2005.

8.4 Évaluation de JUXMEM au sein de DIET

Dans cette section, nous présentons l'évaluation de l'intégration de JUXMEM au sein de l'intergiciel Grid-RPC DIET. Nous avons exécuté notre application motivante Grid-TLSE (voir section 8.2.2) en utilisant JUXMEM-C 0.3 pour la gestion de ses données. Par manque de temps, nous n'avons pas pu réaliser une évaluation complète de l'intégration de JUXMEM

[4]Pour des raisons de lisibilité sur la figure 8.4 $F2$ est différent de $F1$. Toutefois en pratique $F2$ peut éventuellement être confondu avec $F1$.

au sein de l'intergiciel Grid-RPC DIET dans ce contexte. Toutefois, nous avons réalisé une évaluation grâce à une application synthétique reproduisant, en partie, les besoins de Grid-TLSE pour la gestion des données.

Conditions expérimentales et description du test. L'environnement matériel du test est la grille expérimentale Grid'5000 [51], avec un maximum de deux sites participant à cette évaluation (Orsay et Rennes). Dans les deux sites, les machines utilisées sont équipées de biprocesseurs Opteron d'AMD cadencés à 2,0 GHz, munis de 2 Go de mémoire vive et exécutant la version 2.6 du noyau Linux. Compte tenu des valeurs par défaut de la taille des tampons TCP, le débit entre deux machines situées dans ces deux sites est limité à 3,71 Mo/s[5]. Nous avons conservé ce faible débit afin d'émuler un client utilisant une grille de calcul pour exécuter ses requêtes. La latence entre les deux sites est de 4,5 ms.

Par ailleurs, nous utilisons DIET 2.1, JUXMEM-C 0.3, une version modifiée de GoDIET pour le déploiement de DIET configuré avec JUXMEM et le greffon JXTA de l'outil de déploiement ADAGE pour le déploiement de JUXMEM-C (voir section 11.2.3). L'ensemble des exécutifs a été compilé avec gcc 4.0.

Notre environnement logiciel de test est constitué :

1. d'un client, d'un agent maître et de 16 serveurs de calcul DIET ;
2. d'un gestionnaire et d'un fournisseur JUXMEM.

Par ailleurs, le client DIET et tous les serveurs de calcul DIET sont configurés afin d'utiliser JUXMEM, c'est-à-dire qu'ils sont également des clients JUXMEM. Enfin, le client DIET est localisé sur une machine du site de Rennes tandis que les autres processus sont localisés sur des machines du site d'Orsay. Notons que le choix d'un unique fournisseur JUXMEM est justifié par son absence d'impact sur les résultats de cette évaluation particulière. En effet, les serveurs de calcul DIET qui sont des clients JUXMEM dialoguent uniquement avec le même fournisseur (appelé *leader*) quel que soit le degré de réplication de la donnée.

Le code du client exécute 32 appels Grid-RPC asynchrones de la fonction distante sleep, et mesure le temps total t nécessaire pour recevoir toutes les réponses. Chaque requête comporte uniquement deux paramètres en entrée (mode in) : 1) une matrice persistante (de type primitif int) et 2) un entier spécifiant la durée de la fonction sleep. Les paramètres que nous faisons varier sont la durée d d'exécution de la fonction sur les serveurs de calcul (d égal à 5 et 60 secondes), ainsi que la taille m de la matrice (de 4 octets à 64 Mo). Chaque test est réalisé avec DIET configuré avec et sans JUXMEM pour la gestion des données persistantes. L'objectif de ce test est de mesurer le gain apporté par JUXMEM du fait de la prise en charge des données persistantes.

Résultats. La figure 8.5 présente les temps totaux d'exécution t du test précédent, lorsque DIET est configuré avec et sans JUXMEM pour la gestion des données persistantes. La partie gauche de cette figure montre les résultats pour $d = 5$ secondes. Pour $m = 64$ Mo, le temps t d'exécution est de 257 secondes lorsque DIET est configuré sans JUXMEM. Ce temps est ramené à 38 secondes lorsque DIET est configuré avec JUXMEM. Cela un représente un gain de 6,8. Il est principalement dû à la factorisation des échanges des données en une unique communication WAN et 32 communications SAN grâce à l'utilisation de JUXMEM, au lieu

[5]Mesure réalisée par l'outil ttcp.

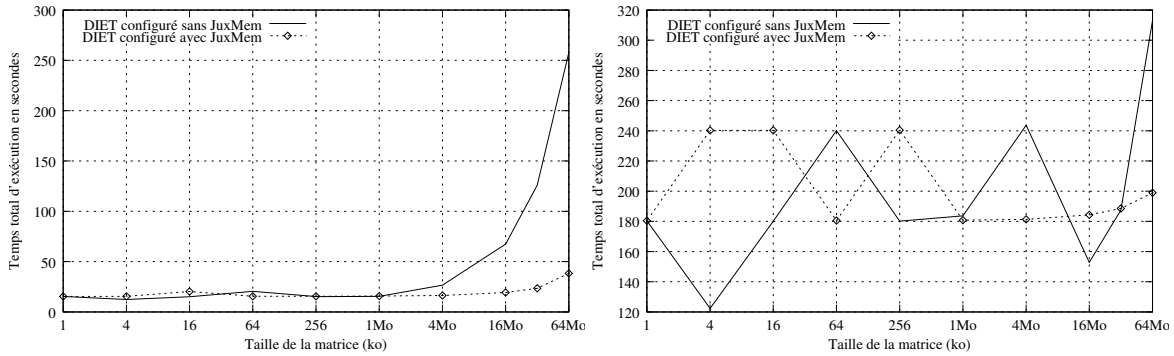

FIG. 8.5 – Temps total d'exécution (t) d'un test avec une matrice persistante de taille variable lorsque DIET est configuré avec et sans JUXMEM ($d = 5$ secondes à gauche et $d = 60$ secondes à droite).

de 32 communications WAN sans JUXMEM. Nous rappelons que le débit du lien en WAN est de 3,71 Mo/s dans cette expérience, alors que le débit des liens SAN utilisés est de 1 Gb/s. Ce gain s'accroît en fonction de la taille des données pour des valeurs de m supérieures à 1 Mo. Par ailleurs, lorsque les tailles de données sont faibles, le surcoût introduit par l'utilisation de JUXMEM est négligeable. En effet, lorsque DIET est configuré avec ou sans JUXMEM, t est égal à 15 secondes. Cette valeur ne correspond pas à la valeur théorique de 10 secondes attendue pour un ordonnancement parfait, compte tenu du nombre de requêtes et de serveurs. Ce chiffre est le résultat d'un mauvais ordonnancement des requêtes par DIET, dans le cas de requêtes asynchrones. L'ordonnancement réalisé consiste dans le placement des 16 premières requêtes sur les 16 serveurs de calcul, suivi par deux séries successives de traitement de 8 requêtes sur 8 serveurs de calcul uniquement. Ce comportement reste pour l'instant inexpliqué.

La partie droite de la figure 8.5 présente le temp total d'exécution t, lorsque $d = 60$ secondes et pour des tailles de matrices m croissantes. Lorsque $m = 64$ Mo, la valeur de t est de 312 secondes sans JUXMEM et 198 secondes avec JUXMEM. Un gain de 1,6 est donc observé. Par rapport aux résultats de l'expérience précédente, ce gain est plus faible. Il s'explique par la prédominance du temps d'exécution de la requête sur les serveurs de calcul par rapport aux temps de transfert des données. Pour des tailles de données inférieures, la valeur de t présente des différences importantes variant de 120 secondes à 240 secondes. Ceci est la conséquence du problème aléatoire d'ordonnancement précédemment évoqué. La valeur de t pour un ordonnancement parfait est de 120 secondes. Dans ce cas, un serveur de calcul traite 2 requêtes. Les valeurs de 180 secondes et 240 secondes correspondent respectivement à au moins un serveur de calcul qui a traité respectivement 3 et 4 requêtes.

Cette expérience montre l'intérêt de l'utilisation de JUXMEM pour la gestion des données persistantes. En effet, un gain sur le temps total d'exécution de notre application synthétique est obtenu lorsque DIET est configuré pour utiliser JUXMEM. Ce gain est proportionnel à la taille des données utilisées. Par ailleurs, rappelons que le développeur d'applications n'a pas à se soucier de la localisation des données. Cela est automatiquement et de manière transparente pris en charge par DIET via son utilisation de JUXMEM.

8.5 Discussion et conclusion

Dans ce chapitre, nous avons introduit dans un premier temps le modèle de programmation Grid-RPC. Puis, nous avons présenté la gestion des données actuellement réalisée dans les intergiciels implémentant ce modèle. Nous avons notamment pu constater que cette gestion présentait des lacunes en termes de fonctionnalités offertes de manière transparente aux applications : localisation, persistance, cohérence, tolérance aux fautes. Or, nous avons montré à travers une application motivante – appelée Grid-TLSE, application de résolution de systèmes linéaires constitués de matrices creuses – que ces fonctionnalités sont nécessaires. L'ensemble des propriétés fournies par le concept de service de partage de données introduit par JUXMEM permet de répondre aux besoins du modèle Grid-RPC pour la gestion des données. Dans un deuxième temps, nous avons donc introduit notre proposition d'utilisation de JUXMEM au sein d'un intergiciel Grid-RPC : DIET. Nous avons présenté DIET puis illustré les modifications nécessaires pour intégrer l'utilisation de JUXMEM par cet intergiciel. Enfin, dans un dernier temps, nous avons évalué notre mise en œuvre par l'intermédiaire d'une application synthétique et de l'application Grid-TLSE.

Cette évaluation a permis de montrer que l'utilisation de JUXMEM au sein de l'intergiciel Grid-RPC DIET est adéquate. Elle permet notamment de répondre aux besoins mentionnés pour la gestion des données (transparence de la localisation des données, cohérence, tolérance aux fautes) tout en permettant un gain en termes de temps total d'exécution d'applications nécessitant une gestion de données persistantes. La gestion des données s'en trouve donc facilitée. Toutefois, des évaluations plus poussées sur des scénarios applicatifs plus complexes sont nécessaires, pour confirmer que l'utilisation de JUXMEM dans le modèle Grid-RPC permet de décharger de manière efficace les développeurs d'applications de la gestion des données. Par ailleurs, notons que l'évaluation que nous avons réalisée est basée sur une version de JUXMEM-C (0.3) dont l'implémentation du rôle de fournisseur est préliminaire et améliorable. Enfin, nous insistons sur le fait que notre proposition est totalement compatible avec les efforts actuellement menés dans le cadre du groupe de travail Grid-RPC du GGF pour standardiser la gestion des données dans ce modèle.

L'utilisation de JUXMEM par DIET que nous avons présentée dans ce chapitre ouvre des perspectives. Tout d'abord, il serait intéressant d'implémenter au niveau des clients JUXMEM un cache des données utilisées afin de conserver des données entre différents appels Grid-RPC. Ainsi, lorsque les serveurs de calculs réutiliseraient la même donnée les performances seraient encore accrues : aucune communication réseau n'est nécessaire si la donnée n'est pas modifiable. Il serait également intéressant d'étudier la prise en compte du placement des données dans les algorithmes d'ordonnancement des calculs. L'objectif est d'améliorer le débit en termes de nombre de requêtes par seconde d'une plate-forme Grid-RPC. Un tel travail nécessite la mise en place dans JUXMEM d'une interface de programmation dont le rôle serait d'évaluer les temps de transfert des données dans un réseau JUXMEM. Ceci permettrait alors de tirer parti de l'implémentation modulable, sous la forme de greffons (en anglais *plugin*), des politiques d'ordonnancement de DIET [17]. Un autre axe de recherche intéressant serait l'étude de l'impact de la gestion des flux de calculs (en anglais *workflow*) sur notre service de partage de données. En effet, dans un tel cadre l'intergiciel Grid-RPC a la possibilité d'indiquer les futures différentes dépendances de données entre les appels Grid-RPC. En conséquence, il serait intéressant d'étudier au sein de JUXMEM différentes politiques de préchargement (en anglais *pre-fetching*) adaptées à la topologie physique des

grilles de calcul et tirant parti d'un cache de données au niveau des clients JUXMEM. L'objectif reste toujours l'amélioration du débit de la plate-forme Grid-RPC utilisée. Enfin, il serait intéressant d'implémenter l'utilisation de JUXMEM dans un autre Grid-RPC afin de : 1) valider le caractère générique de notre proposition pour le modèle Grid-RPC et 2) d'autoriser le partage de données entre intergiciels Grid-RPC et d'en étudier les possibilités.

Chapitre 9

Utilisation et évaluation de JUXMEM dans les modèles composant

Sommaire

Dans ce dernier chapitre de la troisième partie de ce manuscrit, nous présentons l'utilisation et l'évaluation de JUXMEM dans les modèles de programmation à base de composants logiciels. Ainsi, dans un premier temps nous décrivons de manière générale les modèles composant. Puis, nous soulignons leurs carences actuelles dans les modèles d'accès aux données qu'ils fournissent. Dans un second temps, nous introduisons notre proposition pour le partage transparent de données dans les modèles composant. Enfin, nous présentons la mise en œuvre et l'évaluation de notre proposition dans deux modèles composant : le modèle composant CORBA (CCM) et le modèle composant CCA (*Common Component Architecture*). Ce travail a été réalisé en collaboration avec Christian Pérez[1].

[1]Chargé de recherche à l'INRIA au sein du projet PARIS.

9.1 Les modèles de programmation à base de composants logiciels

Dans cette section, nous introduisons de manière générale la programmation à base de composants, sans nous attacher à un modèle en particulier. Par abus de langage et par souci de concision, nous utiliserons le terme *modèles composant* pour faire référence aux modèles de programmation à base de composants.

9.1.1 Objectifs et définitions

Avec l'évolution de la puissance de calcul, les applications scientifiques simulent des phénomènes complexes d'une manière de plus en plus précise. Cela se traduit par un accroissement du code de l'application. Les modèles à base d'objets sont une première étape pour une meilleure modélisation et un meilleur développement de telles applications distribuées, volumineuses et complexes. Toutefois et en dépit de leurs avantages, de tels modèles ne permettent pas de réutiliser simplement des objets qui composent une application. En effet, les relations qui existent entre les différents objets sont enfouies dans le code et ne sont pas clairement définies dans l'interface des objets. Or, la réutilisation de codes existants permet de concevoir et de réaliser des nouvelles applications plus rapidement. Les modèles composant ont pour objectif d'offrir un environnement de développement permettant de faciliter cette réutilisation de codes existants. Ainsi, ils mettent à disposition un modèle de programmation plus adapté aux besoins de conception des applications actuelles.

L'entité de base dans les modèles composant est le *composant logiciel*, appelé plus simplement *composant*. Il existe plusieurs définitions admises pour ce terme, mais la plus courante et la plus reconnue est celle de Clemens Szyperski [169], donnée ci-dessous.

Définition 9.1 : ***composant logiciel*** — Unité de composition qui spécifie contractuellement ses interfaces et qui définit explicitement toutes ses dépendances de contexte. Un composant logiciel peut être déployé indépendamment et est sujet à composition par une tierce partie.

Analysons cette définition. Un composant logiciel peut être vu comme une *boîte noire*, où seules ses interfaces externes sont accessibles par d'autres composants d'une application. En outre, une telle boîte définit le contexte à respecter pour pouvoir l'utiliser dans une application, c'est-à-dire les conditions nécessaires que son environnement d'exécution doit satisfaire. Les fonctionnalités offertes par une telle boîte noire sont implémentées par le *code fonctionnel* du composant, également appelé *code métier*. Par ailleurs, et c'est certainement le point le plus important de la définition, un composant peut être combiné avec d'autres composants pour former une application. Le but est de promouvoir la réutilisation de codes déjà existants. En conséquence, les modèles composant mettent en œuvre les mécanismes nécessaires pour parvenir à développer une telle boîte noire. En particulier, les interfaces des composants ainsi que leurs dépendances sont clairement spécifiées.

À partir de la définition 9.1, nous pouvons également constater que la communication depuis un composant vers le monde extérieur se fait exclusivement par le biais d'interfaces. De telles interfaces sont typées. Dans la terminologie des modèles composant, une interface est disponible via un *port*. Il existe deux types de ports : un composant peut soit *fournir* une interface via un port *provides*, soit *utiliser* une interface via un port *uses*. Deux composants peuvent être donc connectés si et seulement si chacun possède un port de même type, l'un fournissant l'autre utilisant l'interface associée. La figure 9.1 présente deux composants A et

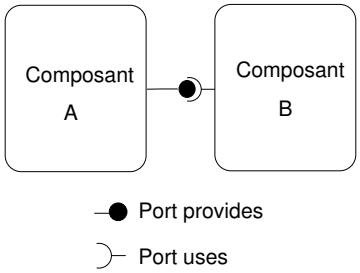

FIG. 9.1 – Illustration de deux composants interconnectés par un port.

B interconnectés par un port *provides* et un port *uses*. Les ports d'un composant sont généralement décrits dans un langage de description d'interface (IDL), qui a pour objectif d'offrir une vision haut niveau d'un ensemble de composants. Notons que les ports n'imposent pas un modèle de communication en particulier : il peut s'agir d'un modèle par appel de méthodes distantes, par envoi d'événements/messages, etc.

De manière classique, dans les langages de description d'interfaces (IDL), les modes des paramètres définissent le propriétaire de la donnée. Pour un paramètre en in, l'appelé ne peut pas réallouer la donnée alors que cela lui est autorisé pour une donnée en inout. Pour un paramètre en out, l'appelé est responsable de l'allocation de la donnée.

9.1.2 Gestion des données dans les modèles composant existants

Les modèles de communication actuellement implémentés pour les ports impliquent que les communications entre composants soient *explicites*. En conséquence, les composants ne peuvent partager des données que par échanges de messages. Un tel modèle de communication nécessite de calculer au préalable quelles parties de la donnée partagée il faut échanger. Ce calcul vient alors se mélanger dans le code fonctionnel du composant, accroissant de manière inutile la complexité du composant. Ainsi, lorsque les structures de données sont complexes et clairsemées en mémoire, ou lorsque les accès aux données sont irréguliers, ce modèle de communication n'est pas adapté ni efficace. Un modèle d'accès transparent aux données tel qu'il a été défini au chapitre 5 semble plus approprié.

Accès aux données via les ports *provides* et *uses*. Les ports *provides* et *uses* d'un composant permettent de simuler l'accès à une donnée partagée. Dans une première approche, la donnée peut être physiquement stockée par le composant qui la fournit, via le port *provides*. Toutefois, dans une telle solution la donnée est centralisée sur un composant, ce qui constitue un goulot d'étranglement pour les performances des accès, particulièrement si leur nombre augmente. En effet, le composant qui héberge la donnée reçoit des messages de tous les autres composants. Par ailleurs, ce composant est une entité critique : si la machine hébergeant le composant est arrêtée, la donnée est perdue. En outre, un code de gestion des accès concurrents doit être mis en œuvre dans le composant partageant la donnée. La figure 9.2 représente un exemple d'une telle approche. Quatre composants (A, B, D et E) accèdent chacun par un port *uses* à une même donnée. Celle-ci est rendue disponible par un port *provides* du composant C.

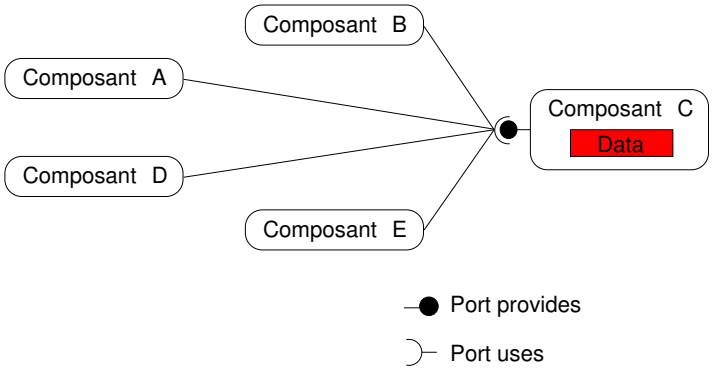

FIG. 9.2 – Illustration de l'approche *centralisée* pour la mise en œuvre du partage de données avec les ports *provides* et *uses*.

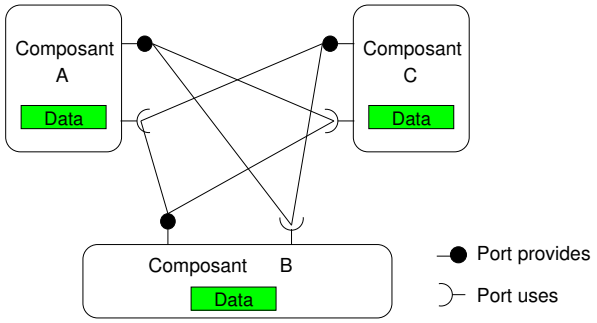

FIG. 9.3 – Illustration de l'approche *distribuée* pour la mise en œuvre du partage de données avec les ports *provides* et *uses*.

Une autre solution consiste à disposer d'une copie de la donnée partagée sur chaque composant qui souhaite y accéder. Toutefois, cette solution nécessite l'introduction de codes de gestion des accès concurrents à la donnée dans le code métier de l'ensemble des composants, notamment pour maintenir la cohérence des copies. En conséquence, la gestion des données est alors *explicite*. Il en résulte une augmentation inutile de la complexité du code. La figure 9.3 représente un exemple d'une telle approche avec 3 composants A, B et C. Chaque composant stocke une copie de la donnée partagée. Par ailleurs, ils disposent chacun d'un port *provides* et *uses* pour la gestion des accès concurrents à la donnée.

Passage de paramètres via les ports *provides* et *uses*. Lors d'un appel à une fonction d'une interface d'un port *provides*, les paramètres sont transférés entre l'appelant et l'appelé. Comme pour les appels Grid-RPC, le mode d'un paramètre (in, inout ou out) spécifie comment les données sont transférées entre deux composants. Dans la plupart des modèles composant, une donnée en mode in ne peut pas être partagée entre l'appelant et l'appelé ou à travers différentes opérations sur différents composants. En effet, lorsque les composants sont situés dans des processus différents, la donnée est passée par valeur. Dans le cas contraire, un pointeur mais en mode *const* est utilisé. Par ailleurs, l'appelant perd généralement le droit d'accéder à la donnée durant l'appel.

Ainsi, pour un paramètre en mode inout, l'appelé est autorisé à détruire et réallouer la donnée. En conséquence, l'appelant va allouer une nouvelle zone mémoire à la réception de la donnée. En conclusion il n'est pas possible à l'appelant d'accéder ou de partager la donnée avec d'autres composants lorsque l'appelé l'utilise.

Nous pouvons constater que les modèles composant actuels ne permettent pas d'offrir aux développeurs d'applications un modèle d'accès aux données, adéquat. Dans la suite de ce chapitre, nous présentons notre proposition pour lever ces deux limites : l'utilisation du modèle d'accès *transparent* aux données fourni par notre service de partage de données JUXMEM. Toutefois, nous présentons d'abord dans la section suivante notre application motivante pour ce travail.

9.1.3 Application motivante : problème à N corps

Nous avons choisi une application pour motiver notre proposition d'enrichir les modèles composant d'un modèle d'accès transparent aux données. Il s'agit d'une application de type *problème à N corps* (en anglais *N-body*) de simulation gravitationnelle de N corps dans un espace tridimensionnel (3D).

Dans une application de type problème à N corps, la position des corps dans l'espace est calculée à chaque étape de la simulation à partir des forces gravitationnelles appliquées sur chaque corps par les autres corps. Cet exemple ayant un but purement illustratif, nous considérons une approche très naïve pour résoudre ce problème. Un corps est représenté par un composant body. Chaque composant accède à un tableau global des positions de tous les corps impliqués dans la simulation. Notons que le nombre de composants body est fixé lors de la phase d'assemblage de l'application. Une telle application permet de mettre en valeur la nécessité d'un modèle d'accès transparent aux données partagées. Les deux approches qu'offrent les ports *provides* et *uses* pour l'accès aux données partagées ne sont donc pas satisfaisantes pour ce type d'application. En effet, compte tenu du nombre potentiellement élevé de corps à simuler et comme nous l'avons vu à la section 9.1.2, l'utilisation d'une approche centralisée engendre une charge importante sur le composant qui stocke le tableau des positions. En conséquence, ce dernier devient un goulot d'étranglement pour les performances de l'application. L'approche distribuée n'a pas ce désavantage, mais nécessite la mise en place de codes de gestion des accès concurrents au tableau pour assurer sa cohérence. Ainsi, la complexité d'un composant body est inutilement accrue avec une inclusion de codes non métier dans l'implémentation des composants.

9.2 Extension des modèles composant pour un partage transparent des données

Comme nous l'avons vu dans la section précédente, il est possible de simuler l'accès à une donnée externe au prix d'une complexité accrue. Dans cette section, nous introduisons notre proposition pour contourner les deux limites précédemment décrites : l'impossibilité de disposer d'un accès transparent aux données ainsi que de partager des paramètres lors de l'invocation d'opérations. Elle est constituée de deux points : la notion de *ports de données* et son utilisation pour le partage de paramètres.

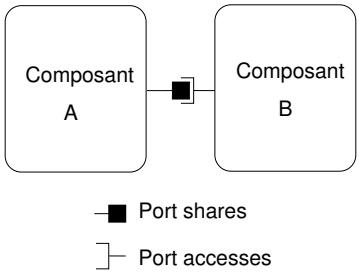

FIG. 9.4 – Illustration de deux composants interconnectés par un port *shares* et port *accesses*.

```
interface data_access {
   float* get_pointer();
   long get_size();
   // Primitives de synchronisation
   void  acquire();
   void  acquire_read();
   void  release();
};
```

```
interface data_shared_port {
    char* get_data_id();
};
```

FIG. 9.5 – Interface d'accès à une donnée offerte au développeur par les ports de données *accesses* (gauche) et *shares* (gauche et droite). La donnée partagée est un tableau de float.

9.2.1 Notion de ports de données

Notre proposition pour réduire la complexité de la gestion des données partagées consiste en un modèle abstrait de *ports de données* (en anglais *data ports*). Ils permettent d'attacher virtuellement une donnée partagée à un composant. Cette proposition vise à enrichir les modèles composant actuels avec les fonctionnalités du modèle d'accès transparent présenté au chapitre 5. Pour ce faire, nous définissons deux types de ports de données. Premièrement, un port de données de type *shares* qui rend disponible aux autres composants une donnée partagée. Deuxièmement, un port de données de type *accesses* qui permet à un composant d'accéder à une donnée exportée par un port de type *shares*. Dans un souci de concision et par abus de langage, nous utiliserons les termes port *shares* et port *accesses* pour désigner respectivement un port de données de type *shares* et un port de donnée de type *accesses*. La figure 9.4 présente deux composants A et B interconnectés par un port *shares* et un port *accesses*.

Comme nous l'avons vu précédemment, les interfaces des ports précisent de manière claire comment les interactions entre composants peuvent avoir lieu. Nous définissons donc deux types d'interfaces : une pour accéder à la donnée, appelée data_access, et une pour rendre la donnée accessible à d'autres composants, appelée data_shared_port. L'interface data_access est disponible sur les deux types de ports. En effet, un composant qui partage une donnée peut vouloir y accéder. La partie gauche de la figure 9.5 présente l'interface de programmation offerte au développeur d'un composant par l'interface data_access. Elle fournit un ensemble de primitives directement inspirées par notre proposition pour l'interface du service de partage de données (voir section 5.1.4). Ainsi, l'interface offre des primitives de synchronisation, telles que acquire, acquire_read et release. La sémantique de ces

```
interface compute {
   void inverse(in matrix &m1, out matrix &m2);
};
```

FIG. 9.6 – IDL de l'opération inverse de l'interface compute.

primitives est identique à celle introduite dans le chapitre 5 pour notre service de partage de données. Elle offre également les primitives get_pointer/get_size qui permettent d'obtenir respectivement un pointeur sur une donnée partagée et sa taille.

L'interface data_shared_port est une interface externe du composant disponible uniquement à partir d'un port *shares*. La partie droite de la figure 9.5 représente cette interface. Elle permet à un composant disposant d'un port *accesses* d'obtenir une référence sur la donnée partagée et fournie par le port *shares*. Pour ce faire, elle contient l'opération get_data_id qui retourne l'identifiant de la donnée. Cet identifiant va alors permettre de projeter une copie de la donnée dans l'espace d'adressage du composant qui souhaite accéder à la donnée via la primitive juxmem_mmap (voir section 5.1.4.1).

Une différence importante existe entre les ports de données et les ports classiques d'un composant. Les ports classiques sont basés sur de l'échange de messages qui déclenche une réaction d'un composant, alors que les ports de données sont passifs. En effet, il n'y a pas de notifications d'événement tel que la modification des données. Dès que le port *accesses* d'un composant A est connecté à un port *shares* d'un composant B, la donnée associée est immédiatement accessible par le composant A. Aucune communication additionnelle entre les deux composants n'est nécessaire. Les accès sont entièrement gérés par l'implémentation du service de partage de données utilisée de manière sous-jacente.

9.2.2 Passage en paramètres de données partagées

Dans la suite de cette section, nous considérons un exemple composé de deux composants : un Client connecté à un Server qui fournit l'interface compute. Cette interface propose une opération appelée inverse (voir figure 9.6). Le composant Client invoque l'opération inverse de cette dernière interface.

Les modèles composant que nous visons définissent un langage de description des interfaces des composants (IDL). Afin de pouvoir partager un paramètre lors de l'appel d'une opération, nous avons introduit une nouvelle notation dans les langages IDL. Cette notation exprime le fait que le paramètre est passé *par référence* et non *par valeur*. A cet effet, le « et » commercial (« & ») apparaît comme un choix évident puisqu'il remplit déjà ce rôle dans le langage C++. La figure 9.6 montre un exemple simple d'utilisation de cette nouvelle notation : l'opération inverse dispose de paramètres en in et out passés par référence.

Les modèles composant imposent que toute communication entrante ou sortante s'effectue via des ports. En conséquence, la notation introduite doit être traduite sous la forme de ports. Notre proposition consiste à utiliser pour de tels paramètres les ports de données introduits à la section précédente. Ainsi, lors de l'appel de l'opération, un port de donnée est associé à chaque paramètre passé par référence. Notons qu'il peut exister une différence dans l'implémentation de l'appelant et de l'appelé. Du côté de l'appelant, les ports de données doivent être explicitement créés par le développeur du composant. Cela est en effet

nécessaire afin de rendre la donnée accessible aux appelés, en particulier pour supporter de multiples invocations simultanées sur des données potentiellement différentes. En revanche du côté de l'appelé, les ports de données peuvent être automatiquement générés par un compilateur IDL, si ce dernier génère déjà le code de gestion des appels entrants.

Illustrons l'utilisation de ce modèle sur l'exemple précédemment introduit. Le listing 9.1 montre comment le composant Client appelle l'opération inverse du composant Server.

Listing 9.1 – Étapes nécessaires pour l'appel de l'opération inverse, depuis le composant Client, en utilisant des données partagées comme paramètres.

```
1 SharesPort *dp1 = createSharesPort("p1", matrix)
2 dp1->associate(ptr, size);
3
4 AccessPort *dp2 = createAccessPort("p2", matrix);
5 to_server->inverse(dp1, dp2);
```

Premièrement, un port de données doit être créé pour chaque paramètre partagé avant tout appel de l'opération. Ainsi, pour le paramètre en in (p1), un port *shares* pour une donnée de type matrix est créé puis associé à la donnée (lignes 1 et 2). Cela permet de rendre la donnée accessible depuis l'extérieur du composant. Pour le paramètre en out (p2), un port *accesses* est créé (ligne 4). Il va permettre d'utiliser la référence de la donnée partagée retournée par l'opération distante inverse. La figure 9.2 montre le code de l'appelé. Le paramètre en in de l'opération inverse a été converti en un port *accesses* (ligne 1) : l'implémentation de cette opération accédera à une donnée déjà existante, via la primitive get_pointeur (ligne 2). Pour le paramètre en out, le résultat produit est associé au port *shares*, par l'intermédiaire de la primitive associate (ligne 7). De manière plus générale, cette étape d'association de la référence à un port *shares* est également nécessaire pour les paramètres en inout.

Listing 9.2 – Implémentation de l'opération inverse disposant de paramètres partagés, sur le composant Server.

```
1 inverse_impl(AccessPort& dpm, SharesPort& dpn) {
2   ptr = dpm.get_pointer();
3   size = dpm.get_size();
4
5   res = f77_inverse(ptr, size);
6
7   dpn->associate(res, size);
8 };
```

Nous tenons à faire remarquer au lecteur que notre notation « & » n'a pas la même sémantique que lors du passage des paramètres. Elle indique uniquement si la donnée est *partagée* ou non. La sémantique de cette notation est donc orthogonale avec la sémantique des modes des paramètres. Pour le mode in, la sémantique est la suivante : l'appelant fournit un accès à une donnée déjà existante. L'appelé peut donc accéder à la donnée sans toutefois avoir le droit de la réallouer, puisqu'elle peut être utilisée par d'autres composants ou par l'appelant. Pour le mode out, l'appelant reçoit une référence sur la donnée partagée et allouée par l'appelé. Enfin, pour le mode inout, l'appelé peut réallouer la donnée en entrée. Plus précisément, comme la donnée peut être partagée et donc accédée par plusieurs composants, il n'est pas possible de simplement la désallouer. Par contre, la référence sur la donnée peut être tout d'abord dissociée du port *shares*, puis une autre référence de données associée à ce même port.

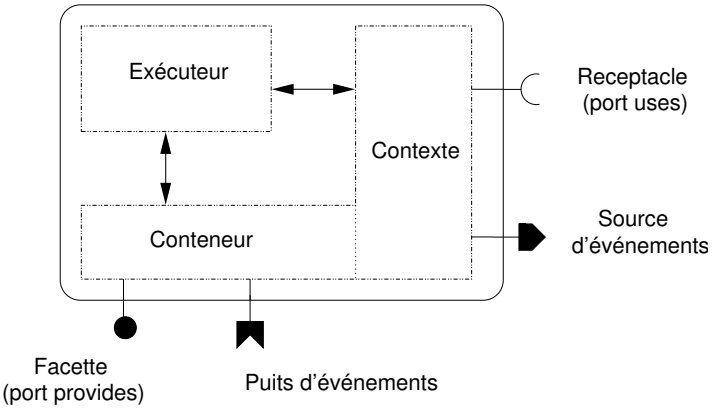

FIG. 9.7 – Vue externe et architecture interne d'un composant CCM.

9.3 Mise en œuvre au sein du modèle composant CCM

Dans cette section, nous décrivons la mise en œuvre et l'évaluation que nous avons réalisées de notre proposition pour l'intégration d'un modèle d'accès transparent aux données dans le modèles composant CCM. Cette projection se base sur JUXMEM pour l'architecture du service de partage de donnée utilisé de manière sous-jacente.

9.3.1 Le modèle composant CCM

CCM (CORBA *Component Model* [136]) fait partie de la troisième version des spécifications de CORBA (*Common Object Request Broker Architecture* [137]) qui est un standard industriel. Les spécifications CCM permettent un déploiement de composants dans un environnement hétérogène et distribué. La figure 9.7 représente la vue externe ainsi que l'architecture interne d'un composant CORBA. Il définit cinq types de ports. Les facettes (ports *provides*) et les réceptacles (ports *uses*) permettent une communication synchrone basée sur le modèle d'appel de méthode à distance. Des communications asynchrones, de type producteur/consommateur, basées sur le transfert de données sont implémentées par des sources et puits d'événements respectivement productrices et consommateurs d'événements. Deux catégories de sources d'événements sont possibles : soit point-à-point entre un composant émetteur (ports *emits*) et un composant consommateur (port *consumes*), soit diffusion à partir d'un composant émetteur (port *publishes*) vers un ensemble de composants consommateurs. En outre, un composant CCM offre un ensemble d'attributs au monde extérieur qui permettent de le configurer. La description des composants et de leurs ports s'effectue en utilisant la version 3 de l'*Interface Definition Language* (IDL3). Les ports sont définis, selon leur type, par les mot-clés provides, uses, etc. La figure 9.8 en présente un exemple.

L'architecture interne (voir figure 9.7) d'un composant CCM est habituellement constituée d'un *exécuteur*, d'un *conteneur* et d'un *contexte*. L'exécuteur contient le code fonctionnel du composant implémenté par le développeur. Le conteneur et le contexte sont générés par un compilateur IDL3. Le conteneur fait le lien entre l'exécuteur et monde extérieur pour les port *provides* et les ports puits d'événements. Le contexte a un rôle similaire mais pour

```
// Définition de l'interface Average
typedef sequence<double> Vector;
interface Average {
  double compute(in Vector v);
};
```

```
// Définition d'un composant A
component aComponent {
  attribute string  name;
  provides  Average avgPort;
  uses      Display dspPort;
};
```

FIG. 9.8 – Exemple de définition d'un composant A par l'intermédiaire de l'IDL3 des spécifications CCM.

les ports *uses* et les ports sources d'événements. Ces liens sont réalisés en fournissant des interfaces bien définies à l'exécuteur.

9.3.2 Proposition de mise en œuvre

Nous avons tenu à respecter deux contraintes pour la mise en œuvre de notre proposition sur le modèle CCM. Premièrement, l'indépendance vis-à-vis de l'implémentation CCM choisie, et cela pour des raisons de portabilité. Deuxièmement, le respect des spécifications CORBA. Bien que cette stratégie puisse ne pas mener à l'implémentation la plus efficace, elle permet de valider notre proposition tout en évitant la complexité d'une modification d'une implémentation de CCM. Par ailleurs, la pérennité de la solution proposée est renforcée puisqu'elle n'est pas liée à une version particulière d'une implémentation de CCM.

Étendre CCM avec les ports de données nécessite de modifier le langage de définition d'interface (IDL3) de CCM. Pour ce faire, notre proposition consiste en l'introduction de deux nouveaux mots-clés dans le langage IDL3 : *shares* et *accesses*. Ce nouveau langage IDL augmenté est appelé IDL3+. Tout en respectant nos contraintes, la spécification IDL3+ est projetée sur le langage IDL3. L'IDL ainsi générée est constituée d'un composant interne qui offre au développeur de composant les opérations fournies par les ports *shares* et *accesses*. Ce composant appartient virtuellement au composant initial et forme un contexte intermédiaire entre l'exécuteur et les autres parties d'un composant CCM standard. La figure 9.9 représente ce composant intermédiaire au sein d'un composant CCM. Toutefois, ce contexte intermédiaire est transparent pour les développeurs de composants. En effet, le composant interne redirige les appels CORBA vers le contexte ainsi que le conteneur. Les appels pour les ports de données eux sont interceptés afin d'appeler les primitives du service de partage de données utilisé de manière sous-jacente.

Notons qu'une interface interne, appelée data_create, est également attachée à un port *shares*. Elle contient pour unique opération la primitive create_data_space, qui permet d'associer une donnée à un port de données. Cette opération est invoquée par l'implémentation du modèle composant quand un port *accesses* est connecté à un port *shares*. Une donnée partagée peut ainsi être créée à l'exécution, via un port *shares*. Enfin, les paramètres de gestion de la donnée, tels que son degré de réplication ou le protocole de cohérence à suivre, sont considérés comme des attributs du composant.

Un exemple de mise en œuvre. La figure 9.3 présente la spécification en IDL3+ d'une application N corps utilisant notre proposition.

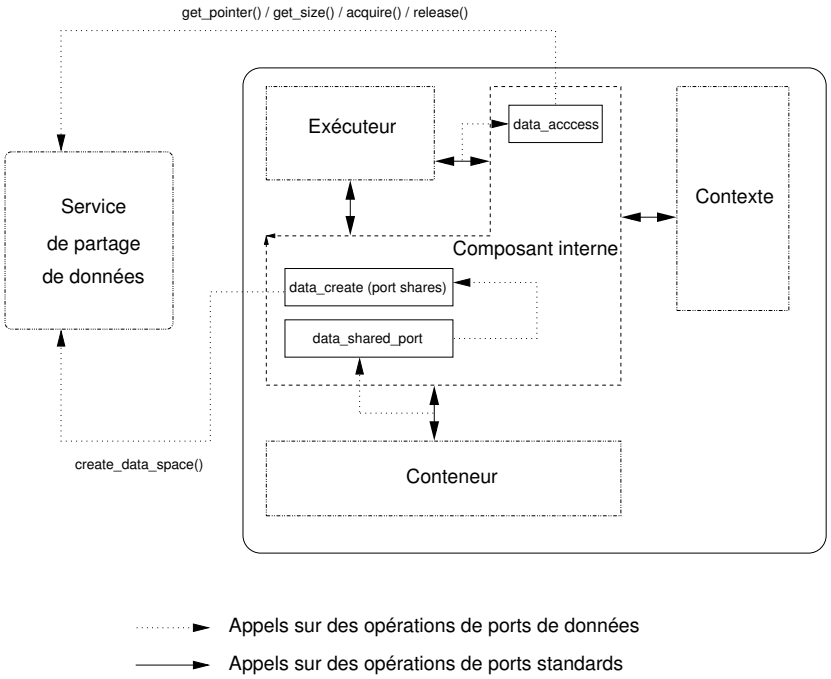

FIG. 9.9 – Architecture interne pour notre proposition de ports de données, implémentée sur CCM.

Listing 9.3 – Un exemple d'IDL3+ pour des ports de données *shares* et *accesses*.

```
1 typedef position float[N][3];
2
3 component sharer { shares position to_bodies; };
4 component body { accesses position from_sharer; };
```

En comparaison avec les types de composants formant cette même application sans la notion de port de données (voir section 9.1.3), un nouveau composant appelé sharer est introduit (ligne 3). Il a pour rôle de rendre disponible le tableau des positions tridimensionnelles des différents corps (ligne 1), via un port *shares* de type position. Les composants body accèdent à ce tableau en déclarant chacun un port *accesses* de type position (ligne 4) qui sera connecté au port *shares* du composant sharer. Toutefois les composants body ont l'illusion d'accéder à une donnée stockée sur le composant sharer. En effet, en réalité le tableau est stocké dans notre service de partage de données.

La figure 9.4 montre comment, dans le code fonctionnel du composant, la donnée est accédée à chaque étape de la simulation. Tout d'abord, un verrou en lecture seule est pris sur le tableau partagé par l'intermédiaire de la primitive acquire_read (ligne 11). Cela permet de récupérer les positions de chaque corps dans l'espace pour calculer, par la suite, la nouvelle position du corps actuellement simulé (ligne 16). Puis, un verrou exclusif est pris sur la position du corps, en utilisant la primitive acquire (ligne 17), avant la mise à jour de sa valeur dans le tableau (ligne 18). Ainsi, la donnée est gérée et accédée comme si elle existait localement sans que le développeur du composant ait à effectuer des transferts explicites de données. L'utilisation des primitives de synchronisation repose directement sur les

primitives offertes par JUXMEM. Notons que pour plus de clarté, le mécanisme de synchronisation mis en œuvre entre chaque étape de la simulation n'est pas représenté. Toutefois, il peut facilement être mis en œuvre par une diffusion d'événements.

Listing 9.4 – Implémentation dans le langage C++ d'une application de type problème à N corps reposant sur nos ports de données.

```
1  class Body_impl : virtual public CCM_body {
2    private:
3      data_access *positions_data_port;
4      int id;
5  };
6
7  void Body_impl::computePosition() {
8   float *data_ptr[];
9   data_ptr = position_data_port.get_pointer();
10
11  positions_data_port.acquire_read();
12  for (i = 0; i < N; i ++) {
13    addToLocalComputation(data_ptr[i]);
14  }
15  positions_data_port.release();
16  // Calcul local ...
17  position_data_port.acquire(data_ptr[id]);
18  updateOurPosition(data_ptr[id]);
19  position_data_port.release();
20 }
```

9.3.3 Évaluation

L'implémentation choisie de la spécification CCM pour notre mise en œuvre est la bibliothèque MicoCCM [228], version 2.3.11. L'implémentation utilisée de JUXMEM est la version 0.2 de JUXMEM-C. Nos choix de mise en œuvre permettent de facilement modifier l'implémentation sous-jacente utilisée pour la gestion des données. Ainsi, nous avons implémenté notre proposition sur un fichier local partagé mais également sur un fichier NFS partagé en plus de l'utilisation de JUXMEM. Toutefois, les performances de cette implémentation n'ont pas été mesurées.

Cependant, nous avons évalué les avantages apportées en termes de programmation du fait de l'utilisation de notre proposition. Ainsi, par rapport à une approche se fondant sur les ports *provides* et *uses* (voir section 9.1.2), notre proposition de ports de données permet de simplifier le développement d'applications de type N corps par exemple. En effet, contrairement à l'approche distribuée pour la mise en œuvre du partage de données avec les ports *provides* et *uses*, aucun code de gestion des données n'est introduit par notre proposition dans le code métier des composants. Par exemple, la cohérence des copies est gérée de manière transparente par JUXMEM et l'accès aux données est réalisé comme si elles étaient stockées localement dans le composant (voir listing 9.4). Par ailleurs et contrairement à l'approche centralisée cette fois-ci, aucun composant ne constitue dans notre proposition ni un goulot d'étranglement pour les performances des accès aux données ni une entité critique pour l'accès aux données.

9.4 Discussion et conclusion

Dans ce chapitre, nous avons dans un premier temps introduit les modèles de programmation à base de composants. Ces modèles ont classiquement comme objectif de réduire la complexité grandissante des applications scientifiques. Pour ce faire, ils prônent la réutilisation de codes et l'isolement du code métier par rapport aux divers codes de gestion (sécurité, persistance, etc.). Toutefois, nous avons montré que cet objectif n'est pas atteint pour la gestion des données. En effet, nous avons notamment identifié deux limites : un modèle d'accès aux données qui laisse à la charge du développeur de composants les transferts des données, et l'impossibilité de partager des paramètres d'opérations distantes. Dans un deuxième temps, nous avons présenté notre proposition pour pallier ces limites. Elle consiste en la définition, de manière abstraite, de la notion de port d'accès aux données partagées. Nous avons également introduit une notation (« & ») permettant de partager des paramètres pendant l'invocation d'opérations distantes. Enfin, dans un dernier temps nous avons présenté la mise en œuvre de notre proposition de ports de données sur le modèle CCM. Celle-ci a été implémentée, en se basant sur JUXMEM pour la gestion des données partagées et sur MicoCCM pour l'implémentation du modèle CCM. Cette mise en œuvre a permis de fournir un deuxième cas d'utilisation pour valider l'implémentation de JUXMEM. Notre proposition d'implémentation de la notation « & » a été instanciée sur le modèle CCA (mais non concrètement réalisée). Elle est décrite à l'annexe A.3.

Toutefois faute de temps, nous n'avons pas réalisé d'évaluation des performances de notre proposition. En revanche, nous avons évalué le bénéfice apporté par notre concept de port de données sur la facilité de programmation d'applications nécessitant un partage de données. Notre discussion montre que contrairement à une mise en œuvre reposant sur les ports *provides* et *uses*, nos ports de données offrent une transparence de gestion des données grâce à l'utilisation d'un service de partage de données tel que JUXMEM (voir chapitre 5). Ainsi, le programmeur a l'illusion que la donnée est localisée dans un composant même si elle est physiquement stockée ailleurs. La gestion des accès concurrents et celle des synchronisations ne sont pas mélangées avec le code métier des composants. De plus, ces gestions ne sont pas à la charge des implémentations des modèles composant, permettant d'éviter de modifier ceux-ci. Ainsi, notre proposition permet aux modèles composant de faire encore un pas en avant dans la réduction de la complexité de la construction d'une application. Par ailleurs, elle permet d'éviter la présence d'une entité critique à la fois pour l'accès aux données ainsi que pour les performances de ses accès. En outre, l'utilisation des ports de données permet de réaliser de manière transparente le partage de données passées en paramètres de méthode distante. Enfin, les programmeurs n'ont pas à se soucier de l'implémentation utilisée pour partager une donnée entre composants : c'est la responsabilité de l'implémentation du modèle composant. Notre solution permet donc non seulement d'utiliser les implémentations de JUXMEM mais également d'utiliser d'autres mécanismes, en fonction de l'infrastructure physique visée.

Du côté des modèles de composant, notre proposition est générique comme l'a démontré l'implémentation sur le modèle composant CCM et sa projection sur les spécifications des interfaces de programmation du modèle CCA (bien que concrètement non validée). Ainsi, nous pensons que d'autres modèles composant, comme Fractal [44], peuvent bénéficier de nos propositions pour intégrer un modèle d'accès transparent aux données.

Quatrième partie

Travaux annexes : adaptation de JXTA pour les grilles de calcul

---

Dans cette quatrième partie du document, nous présentons deux contributions annexes. Elles sont orthogonales à notre principale contribution, qui a été décrite jusqu'à présent. Dans notre contexte, l'objectif de ces contributions est de faciliter l'utilisation de JXTA sur les grilles de calcul et de l'optimiser pour concevoir, mettre au point et implémenter JUXMEM. Nous tenons à souligner que ces contributions relatives à JXTA ne sont pas spécifiques à JUXMEM. Le choix de JXTA est guidé par son utilisation au sein de JUXMEM, toutefois nos motivations et notre méthodologie pour ce travail sont indépendantes de la mise en œuvre au sein de JXTA. Ainsi, nos travaux sont susceptibles de faciliter d'autres recherches sur l'utilisation de JXTA et du modèles P2P sur les grilles de calcul. Ces contributions portent sur : 1) l'évaluation et l'adaptation des couches de communication de JXTA aux réseaux disponibles au sein des grilles de calcul et 2) un langage de description d'applications basées sur JXTA qui a permis l'évaluation à grande échelle d'une partie de la spécification JXTA.

Chapitre 10

Optimisation des performances des couches de communication de JXTA

Sommaire

Ce chapitre porte sur notre première contribution : l'évaluation et l'optimisation des performances des couches de communication de JXTA. Notre motivation pour ce travail d'optimisation des débits et des latences des couches de communication de JXTA est l'amélioration des performances de JUXMEM pour la lecture et l'écriture des données. Nous présentons tout d'abord le fonctionnement interne des couches de communication de JXTA. Puis, nous introduisons nos motivations ainsi que notre méthodologie de test pour la réalisation de ce travail et nous détaillons les optimisations que nous avons réalisées. Nous présentons également l'impact de ces optimisations sur les performances des couches de communication de JUXMEM. Ensuite, nous décrivons notre portage de JXTA-C au-dessus de PadicoTM, étape supplémentaire pour la pleine et transparente utilisation des capacités des réseaux haute performance généralement disponibles dans les grilles de calcul. Enfin, nous concluons par une discussion de nos travaux.

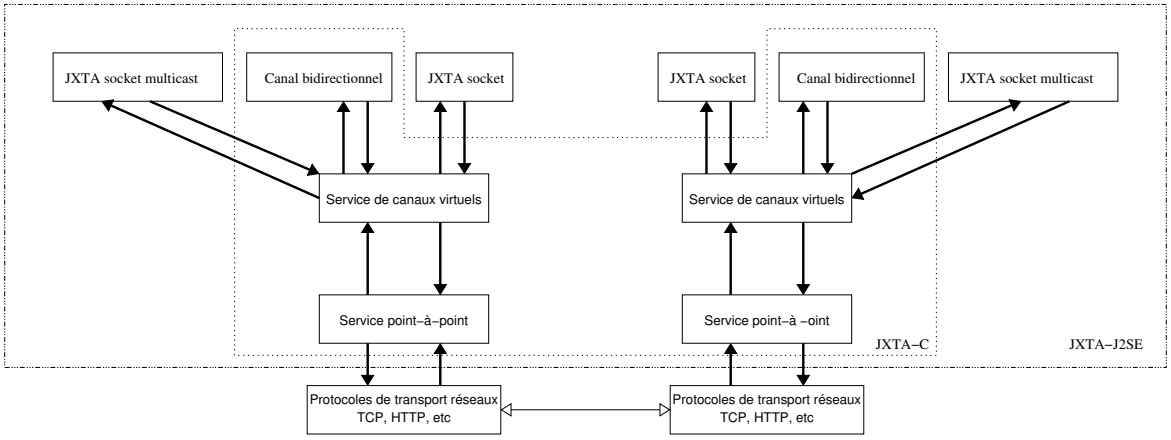

FIG. 10.1 – Pile des couches de communication de JXTA.

10.1 Présentation des couches de communication de JXTA

Dans cette section, nous présentons tout d'abord de manière générale les couches de communication de JXTA ainsi que le format d'un message JXTA. Puis, nous décrivons le fonctionnement interne des services point-à-point et de canaux virtuels, tels qu'ils sont implémentés dans JXTA-C et JXTA-J2SE. Le choix de ce sous-ensemble parmi les couches de communication de JXTA est justifié par leur utilisation dans JUXMEM.

10.1.1 Présentation générale

La spécification JXTA définit deux mécanismes pour la communication entre les pairs : le service point-à-point (*endpoint service*) et le service de canaux virtuels (*pipe service*). Chaque service offre un niveau d'abstraction différent. Le service point-à-point est la couche la plus basse sur laquelle s'appuie le service des canaux virtuels. Comme nous l'avons vu au chapitre 6, ce dernier service propose deux types de canaux virtuels : unidirectionnel et de diffusion. Par ailleurs, les deux implémentations de référence de JXTA introduisent d'autres couches de communication qui ne font pas partie des spécifications. Ainsi, JXTA-C et JXTA-J2SE fournissent des canaux virtuels bidirectionnels. JXTA-J2SE offre au-dessus des canaux virtuels unidirectionnels une interface de communication similaire aux *sockets* BSD : les *sockets* JXTA (JXTA *sockets*). Enfin, les canaux virtuels de diffusion servent également pour la construction d'une couche de communication supplémentaire : les *sockets* multicast JXTA, qui offrent une interface similaire aux *sockets* multicast. Cette dernière couche est également uniquement disponible dans JXTA-J2SE. La figure 10.1 représente cette pile des différentes couches de communication de JXTA.

Au niveau le plus bas, l'unité élémentaire d'échange d'informations entre les pairs est un *message* JXTA . Un tel message JXTA est défini par la spécification comme une séquence non ordonnée de noms et de valeurs typées appelés *éléments* [222]. La spécification JXTA définit un élément comme un objet contenant un espace de nommage (en anglais *namespace*), un nom optionnel, un type optionnel, une signature optionnelle et un contenu. Un nombre non défini d'éléments peuvent être ajoutés par chacune des couches de communications de JXTA

```
tcp://131.254.202.39:15000
```

Figure 10.2 – Exemple d'un élément *adresse source* inclus dans un message JXTA.

et par les couches applicatives. Lors de la réception d'un message, ces éléments doivent être retirés par les services les ayant ajoutés. Par exemple, tous les protocoles spécifiés par JXTA utilisent des éléments XML pour le format de leurs requêtes et de leurs réponses.

JXTA précise également le format réseau d'un message JXTA et de ses en-têtes, qu'il est recommandé d'implémenter. Toutefois, le respect de ce format est optionnel. Deux représentations sont ainsi décrites : binaire, quand par exemple le protocole de transport TCP est disponible, et XML. Le format binaire est conçu de telle manière que toutes ses composantes sont d'une taille déclarée, aucune analyse lexicale n'est nécessaire pour déterminer leur longueur. Ainsi, aucun surcoût n'est rajouté aux performances de lecture des différents éléments constituant un message JXTA.

Enfin, la spécification JXTA détaille également les étapes d'une connexion entre deux pairs JXTA qu'il est recommandé d'implémenter. Elle est constituée de quatre phases : ouverture de la connexion, échange de messages de bienvenue, échange des messages applicatifs, fermeture de la connexion. Les messages de bienvenue permettent principalement d'identifier le pair contacté.

10.1.2 La couche la plus basse : le service point-à-point

Le service point-à-point abstrait les protocoles réseaux (appelés *point d'accès*) qui sont disponibles sur un pair pour échanger des données avec un autre pair. Les protocoles de transport actuellement supportés dans les deux implémentations de JXTA sont TCP et HTTP. Toutefois et quel que soit le protocole de transport utilisé, les communications du service point-à-point sont asynchrones, unidirectionnelles et non fiables. En effet, un message est susceptible de passer par plusieurs pairs avant d'atteindre sa destination. Or, ces pairs intermédiaires peuvent ignorer le message, par exemple s'ils sont surchargés.

L'information nécessaire pour qu'un pair puisse envoyer un message à un autre pair est l'*adresse du point d'accès* du pair de destination. L'adresse d'un point d'accès n'est autre que l'identifiant du pair sur le réseau virtuel JXTA. Pour résoudre un point d'accès, le service point-à-point utilise le protocole de routage (voir section 6.1.3) afin de trouver une route vers le pair de destination, en utilisant les protocoles de transport disponibles. Le service point-à-point utilise deux éléments appelés *adresse source* (EndpointSourceAddress dans le code) et *adresse de destination* (EndpointDestinationAddress dans le code) inclus dans chaque message. Ils ne font pas partie de la spécification JXTA. Les figures 10.2 et 10.3 présentent respectivement un exemple de ces éléments d'adresse source et d'adresse de destination qui permettent d'identifier les pairs d'origine et de destination respectivement. Ces deux éléments contiennent au minimum l'adresse TCP/IP du pair lorsque le protocole TCP est utilisé. En outre, l'élément adresse de destination spécifie le nom du service de groupe en charge du traitement du message lors de sa réception par le pair de destination. Sur la figure 10.3 le service destinataire est le service de routage du groupe *net*, inclus dans le service point-à-point dans cette implémentation, ici JXTA-J2SE.

```
tcp://131.254.202.38:15000/EndpointService:jxta-NetGroup/EndpointRouter
```

Figure 10.3 – Exemple d'un élément *adresse de destination* inclus dans un message JXTA.

10.1.3 Le cœur des communications : le service de canaux virtuels

Par rapport au service point-à-point et comme nous l'avons vu dans le chapitre 6, le service de canaux virtuels permet de s'abstraire de l'identité des pairs impliqués dans des communications par l'utilisation de *canaux virtuels*. Comme les pairs, chaque canal virtuel a un identifiant unique sur le réseau virtuel JXTA. Avant tout transfert, chaque extrémité du canal virtuel est résolue sur un point d'accès, et le service point-à-point est utilisé pour gérer les détails de transmission entre les pairs. La résolution est faite seulement une fois pour chaque pair et est vérifiée toutes les 20 minutes dans, par exemple, JXTA-J2SE. Le service de canaux virtuels fournit donc l'illusion d'un point d'accès indépendant de toute localisation des pairs, comme décrit par les spécifications de JXTA.

De manière similaire au service point-à-point, les communications via ce service de canaux virtuels sont asynchrones, unidirectionnelles et non fiables. Il existe différentes qualités de service pour un canal virtuel : unidirectionnel asynchrone, synchrone requête-réponse, de type flux (en anglais *streaming*), etc. Le canal virtuel unidirectionnel, asynchrone et non fiable est le seul requis à être disponible sur toutes les implémentations se conformant à la spécification JXTA. Les canaux virtuels offrent deux types de communication : soit point-à-point, soit de propagation. En utilisant ces derniers types de canaux virtuels, un pair peut diffuser un même message à n pairs. En outre, dans le mode point-à-point, il est possible d'échanger des messages chiffrés par l'intermédiaire de canaux virtuels sécurisés via TLS. Dans ce chapitre nous nous focalisons uniquement sur les canaux virtuels unidirectionnels en raison de leur utilisation pour l'implémentation des communications au sein de JUXMEM (voir section 7.4). Ils sont par ailleurs utilisés dans la mise en œuvre de toutes les couches de communication supérieures de JXTA.

En termes de composition des messages, le nom du service contenu dans l'élément adresse de destination est le service de routage point-à-point. L'élément appelé *routeur* (EndpointRouterMsg dans le code) est ajouté dans chaque message émis par le service de canaux virtuels. La figure 10.4 montre un exemple d'un tel élément qui se présente sous la forme d'un document XML. Il est traité par le service de routage afin de transmettre le message au service (JXTA ou applicatif) auquel il est destiné. Pour ce faire, cet élément dispose du champ Dest qui définit le service et un paramètre spécifique à ce service à invoquer sur l'identité du pair indiqué. Sur l'exemple, le service des canaux virtuels est le destinataire et le paramètre spécifique correspond à identifiant d'un canal virtuel. Les champs Src et LastHop correspondent à l'identifiant du pair qui a émis le message et à l'identifiant du dernier pair par lequel le message a été retransmis. Sur la figure 10.4 ces deux champs sont identiques, la connexion entre le pair émetteur et le pair de réception étant directe. Enfin, les champs Fwd et Rvs décrivent respectivement une liste non ordonnée de pairs en guise de route vers le pair de destination et dans le sens opposé. Les spécifications précisent que l'élément routeur est utilisé uniquement lorsque des pairs ne sont pas directement connectés. Toutefois, dans les implémentations de JXTA[1] il est utilisé même dans le cas où les pairs sont directement

[1] JXTA-C et JXTA-J2SE.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:ERM>
<jxta:ERM xmlns:jxta="http://jxta.org">
        <Src>
                jxta://uuid-5961626164616261 4A7874615032503309EBF5B2A7334E99003
        </Src>
        <Dest>
                jxta://uuid-5961626164616261 4A78746150325033ACB98E2B3969491BBD5
                CA9BB02D49D0B03/PipeService/urn:jxta:uuid-5961626164616261 4E504
                720503250336AE65D186B0646E191FDEEC5B02ADB3204
        </Dest>
        <LastHop>
                jxta://uuid-5961626164616261 4A7874615032503309EBF5B2A7334E99003
        </LastHop>
        <Fwd/>
        <Rvs/>
</jxta:ERM>
```

Figure 10.4 – Exemple d'un élément *routeur* inclus dans un message JXTA.

connectés.

10.2 Optimisation des performances pour les environnements de type grille

Dans cette section, nous présentons les optimisations que nous avons réalisées dans les couches de communication de JXTA. Notre objectif est de mieux tirer parti du potentiel offert par les réseaux haute performance disponibles dans les grilles de calcul. Nous introduisons donc tout d'abord nos motivations pour ce travail ainsi que notre méthodologie de test. Puis, nous décrivons ces optimisations ainsi que leurs impacts sur les performances en termes de débit et de latence, par l'intermédiaire d'un test bidirectionnel de bande passante. Enfin, nous les évaluons au sein de la plate-forme haute performance de communication pour grilles PadicoTM, qui permet une utilisation transparente des capacités des réseaux haute performance disponibles sur ces infrastructures.

10.2.1 Motivations et méthodologie de test

La taille des grilles de calcul augmentant, celles-ci expriment pour leur gestion un besoin grandissant de mécanismes distribués, flexibles et passant à l'échelle. À travers les différents systèmes pair-à-pair (P2P) qui existent, le modèle P2P a prouvé son efficacité dans ce domaine. L'idée d'utiliser une approche P2P pour, par exemple, découvrir les ressources d'une grille de calcul est alors apparue naturellement. Toutefois, utiliser des systèmes P2P sur des grilles de calcul n'est pas trivial et reste actuellement un défi. En effet, les applications s'exécutant sur ce type d'infrastructures ont généralement des contraintes de performances ce qui n'est pas, en règle générale, le cas des systèmes P2P. Dans un tel contexte et outre les mécanismes de découverte, un autre problème important est le transfert efficace de données.

Utiliser des bibliothèques P2P pour de tels transferts nécessite de pleinement utiliser les capacités des réseaux disponibles sur ce type d'infrastructures : réseaux locaux à faible latence et haut débit (SAN) et réseaux longue distance à haut débit (WAN). En effet et comme nous l'avons vu dans le chapitre 2, une grille de calcul est généralement construite comme une fédération distribuée de grappes de machines. Les SAN, tels que Gigabit Ethernet, Myrinet ou Infiniband, fournissent une bande passante allant de 1 Gb/s (Gigabit Ethernet) à 10 Gb/s (Myrinet) voire 20 Gb/s (Infiniband) et ont une latence de quelques microsecondes. Par exemple le réseau Myrinet Myri-10G offre une latence de 2,5 μs [230]. Les WAN eux offrent un débit similaire mais souffrent d'une plus grande latence : typiquement d'un ordre de grandeur entre 10 ms et 200 ms, selon l'échelle de la grille considérée (nationale, multinationale, continentale, intercontinentale). Cela constitue un scénario de déploiement inhabituel pour les systèmes P2P, qui en général visent plutôt des machines reliées à Internet par liaisons bas débit (quelques Mb/s) avec une forte latence (jusqu'à 300 ms), telles que les connexions ADSL. En conséquence, il est important de s'interroger sur l'adéquation des communications P2P dans un tel contexte, mais également sur la possibilité de les adapter afin de mieux tirer parti du potentiel de ces réseaux haute performance, notamment les SAN.

Pour évaluer les performances des couches de communication de JXTA, nous avons choisi un test bidirectionnel de bande passante, également appelé test de *ping-pong*. Ce choix est justifié par sa large utilisation pour mesurer les performances des protocoles réseaux, ainsi que pour sa capacité à fournir des informations pertinentes sur les performances comme la bande passante et la latence. Chaque test est constitué d'une suite de mesures pour des données applicatives variant d'une taille de 1 octet à 16 Mo. Toutes les mesures sont prises au niveau applicatif et sont calculées sur la base de la moyenne de cinq mesures de 100 échanges consécutifs entre deux pairs. Lorsque JXTA-J2SE est testé, 1 000 messages supplémentaires sont échangés afin de minimiser les perturbations introduites par le compilateur *Just-In-Time* (JIT). Toutes les implémentations testées ont été configurées pour utiliser TCP comme protocole de transport entre les pairs. Une autre métrique que nous avons utilisée est la mesure de l'efficacité d'un protocole. L'efficacité r d'un protocole se définit comme le ratio entre la taille de la donnée applicative à envoyer et la taille totale du message réellement transmis :

$$r = \frac{\text{taille du message applicatif}}{\text{taille totale du message}}$$

Les résultats fournis pour cette métrique sont basés sur l'utilisation du couple d'analyseurs de protocoles réseaux : `tcpdump` et `ethereal`. Lors des tests de JXTA-J2SE, la machine virtuelle Java (JVM) utilisée par défaut est celle de Sun Microsystems, version 1.4.2_01-b06, avec les options suivantes : `-server -Xms256M -Xmx256M`. Enfin, les tests menés sur JXTA-C ont été compilés avec `gcc` 4.0 (niveau d'optimisation `-02`).

Nous avons exécuté ce test bidirectionnel de bande passante sur deux types de réseaux, dont les caractéristiques sont décrites ci-dessous.

Tests SAN. Les réseaux utilisés pour les tests SAN sont Gigabit Ethernet et Myri-10G (driver MX). Lorsque le réseau est Myri-10G, les machines sont équipées des biprocesseurs Opteron d'AMD cadencés à 2,0 GHz, munis de 2 Go de mémoire vive et exécutant la version 2.6 du noyau Linux. Pour le réseau Gigabit Ethernet, les machines sont des biprocesseurs Opteron d'AMD à 2,2 GHz, également équipés de 1 Go de mémoire vive et exécutant également la version 2.6 du noyau Linux. Les connexions réseaux entre les

Implémentation	JXTA-J2SE 2.3.2		JXTA-C 2.2	
Métrique	Débit	Latence	Débit	Latence
Canal unidirectionnel	80 Mo/s (16 Mo)	711 μs	96 Mo/s (0,25 Mo)	294 μs
Service point-à-point	80 Mo/s (4 Mo)	294 μs[a]	102 Mo/s (0,25 Mo)	149 μs
Langage	Java		C	
BSD *sockets*	111 Mo/s (4 Mo)	46 μs	116,5 Mo/s (4 Mo)	39 μs

[a] 268 μs avec la JVM 1.5 de Sun.

TAB. 10.1 – Performances initiales en termes de débit et de latence de JXTA-J2SE 2.3.2 et JXTA-C 2.2 sur un réseau SAN Gigabit Ethernet pour chaque couche de communication étudiée, comparées aux performances des *sockets* BSD. Pour les débits, les chiffres entre parenthèses correspondent à la taille du message pour laquelle le débit maximal est atteint.

machines d'une même grappe utilisant un SAN étant directes, des communications directes entre les pairs ont été configurées.

Tests WAN. La plate-forme utilisée pour les tests en WAN est la grille expérimentale Grid'5000 [51], plate-forme présentée au chapitre 2. Les tests ont été réalisés entre les sites de Rennes et de Toulouse. Dans chaque site, les machines sont équipées des biprocesseurs Opteron d'AMD à 2,2 GHz, également équipés de 1 Go de mémoire vive et exécutant la version 2.6 du noyau Linux. Lors des tests[2], les deux sites étaient interconnectés par une liaison à 1 Gb/s, avec une latence moyenne de 11,2 ms. Les communications entre les noeuds des différents sites Grid'5000 étant directes, des communications directes entre les pairs ont été configurées.

10.2.2 Optimisation des protocoles JXTA et évaluations

Évaluations initiales. Le tableau 10.1 résume les performances initiales en termes de débit et de latence des couches de communication de JXTA-J2SE 2.3.2 et JXTA-C 2.2 sur un réseau Gigabit Ethernet, en SAN. Elles représentent les performances initiales de ces intergiciels telles que nous les avons évaluées. Les courbes correspondantes sont disponibles à l'annexe A.2.1. Comparées aux performances des *sockets* BSD, nous pouvons constater que les performances en termes de latence ne sont pas bonnes : entre 149 μs et 711 μs selon les couches et le langage d'implémentation contre entre 39 μs et 46 μ pour les *sockets* BSD. En outre, les débits présentés de JXTA-J2SE (80 Mo/s) ne sont atteints que pour une taille de messages t importante : pour les canaux unidirectionnels t est égal à 16 Mo et pour le service point-à-point la valeur de t est 4 Mo. Les courbes de débits de JXTA-C sont plus proches des performances des *sockets* BSD : elle accuse au mieux un retard de seulement 14,5 Mo/s, contre 31 Mo/s au mieux pour JXTA-J2SE. Ces débits sont rapidement atteints par JXTA-C, c'est-à-dire pour de faibles tailles de messages ($t = 256$ ko). Cependant, au-dessus de cette taille les performances stagnent, contrairement aux performances des *sockets* BSD. Ces mauvais résultats sont principalement dus à un mécanisme coûteux de limitation de la taille de message émis pour JXTA-J2SE et une couche de communication relativement récente qui est en cours de développement pour JXTA-C.

[2] Configuration de la grille expérimentale Grid'5000 en février 2005. Depuis la liaison a un débit de 10 Gb/s.

Les performances en WAN de JXTA-J2SE et JXTA-C sont comparables aux performances des *sockets* : quasi-saturation du lien à 1 Gb/s utilisé pour ces tests (débit de 98 Mo/s comparé à 104 M pour une *socket* C). De tels débits peuvent être atteints sous réserve d'une configuration adéquate de la taille des tampons TCP pour être en accord avec le produit *débit* $*$ *délai* des caractéristiques de la liaison. Dans notre cas, cela donne une valeur pour la taille des tampons TCP de 1,9 Mo/s[3]. Enfin, la taille totale d'un message JXTA contenant 1 octet applicatif est de 961 octets pour un canal unidirectionnel ($r = 0,1$ %) et de 233 octets pour le service point-à-point ($r = 0,4$ %)[4]. Une présentation plus détaillée de ces résultats sur différentes versions de JXTA-J2SE et JXTA-C, ainsi que des évaluations sur des réseaux Fast Ethernet (100 Mbits) et Myrinet-2000[5], sont disponibles dans [5, 6, 12].

Les performances des couches de communication de JXTA-J2SE et JXTA-C en WAN ne pénalisent pas les transferts de données entre les différents sites d'une grille. En revanche, les performances des couches de communications de JXTA-J2SE et JXTA-C en SAN limitent l'utilisation efficace des capacités de ces réseaux lors des transferts de données dans JUXMEM. L'architecture des grilles de calcul est hiérarchique : faible latence au sein d'un SAN et forte latence en WAN. Les protocoles de cohérence utilisés tiennent compte de cette caractéristique. Ainsi, ils favorisent les messages échangés dans un SAN, en privilégiant par exemple le passage du verrou associé à une donnée entre processus s'exécutant sur ce même SAN. De la même manière, lors du premier accès en lecture à une donnée au sein d'un SAN, un groupe local de gestion de cette donnée est créé afin d'éviter de coûteuses communications en WAN. L'appartenance d'un ensemble de processus à un même SAN est définie par l'utilisation du même groupe cluster, tel qu'il a été introduit dans l'architecture de JUXMEM (voir section 5.2).

Optimisations. Une évaluation du code de l'implémentation des ces couches de communication de JXTA nous a permis de constater que l'origine des ces importantes latences sur des SAN est triple. Premièrement, l'efficacité r des couches de communication de JXTA est faible. Deuxièmement, le code du chemin critique pour l'envoi et la réception d'un message n'est pas optimisé pour minimiser le temps de traitement. Troisièmement, une optimisation peut être réalisée dans le cas particulier où des communications directes entre les pairs sont possibles. Or, dans le contexte classique des systèmes P2P ce cas est une exception, mais il ne l'est plus dans le cas des grilles de calcul. Il devient même un cas standard. Nous avons donc jugé utile d'optimiser les couches de communication de JXTA pour une utilisation dans le contexte des grilles, l'objectif étant d'améliorer les performances des transferts de données au sein de JUXMEM. Pour la réalisation et la validation de nos optimisations, nous nous sommes concentrés sur le couple JXTA-C et JUXMEM-C. Toutefois, les mauvaises performances de JXTA-J2SE ont des origines similaires aux mauvaises performances de JXTA-C. Ainsi, les solutions que nous apportons sont applicables de manière similaire à JXTA-J2SE. Une description, plus détaillée, des optimisations proposées et implémentées est donnée ci-dessous.

Condensation et mise en cache d'éléments. Le service point-à-point ajoute deux éléments à chaque message JXTA émis : adresse source et adresse de destination (voir section 10.1.2). L'information apportée par l'élément adresse source est incluse dans celle

[3]Obtenue par multiplication de la valeur théorique par un facteur arbitraire de 1,2.
[4]Pour TCP la taille totale d'un message de 1 octet est de 161 octets, soit $r = 0,6$ %.
[5]Via la fonctionnalité d'émulation de la couche TCP/IP du driver GM de Myrinet.

fournie par les messages de bienvenue, échangés avec le pair de destination lors de l'établissement de la connexion. Cette optimisation consiste donc à supprimer cet élément. Une partie de l'information apportée par l'élément adresse de destination, l'adresse TCP/IP du pair de destination, est également incluse dans les messages de bienvenue. Cette autre optimisation consiste en la réduction de l'élément adresse de destination à la seule information utile : le nom du service en charge du traitement du message lors de sa réception. Ces optimisations s'apparentent à la technique classique de mise en cache des en-têtes (en anglais *header caching*), largement utilisée par différentes implémentations de protocoles réseaux.

Connexions directes. L'élément routeur peut être supprimé lorsqu'une connexion directe entre les pairs peut être établie. En effet, cet élément XML n'est pas nécessaire dans ce cas. En outre, lorsque l'adresse de destination est soit fournie sous la forme TCP/IP, soit présente dans une table de hachage du pair, ayant pour clé l'identifiant du pair de destination et comme valeur l'adresse TCP/IP associée, le protocole de routage n'est plus utilisé. Cette optimisation permet d'éviter le traitement XML associé de l'analyse lexicale de l'élément routeur, ainsi que la traversée de la couche logicielle implémentant le protocole de routage.

Mode zéro-copie. Cette optimisation consiste à enregistrer une fonction de rappel associée à un espace de nommage applicatif. Chaque élément d'un message JXTA peut en effet définir son propre espace de nommage (voir section 10.1.1). Chaque fois qu'un message JXTA reçu contient un espace de nommage enregistré, la fonction de rappel correspondante est appelée. Cette fonction doit retourner une adresse mémoire valide et d'une taille suffisante pour stocker la valeur de l'élément lu. Pour ce faire elle dispose du nom de l'élément. La donnée pourra alors être stockée directement dans la zone mémoire allouée par l'utilisateur au niveau applicatif. Cette optimisation est utilisée au sein de JXTA pour limiter le nombre de copies mémoire, et éventuellement atteindre un transfert de données en mode zéro-copie lorsque l'encodage XDR n'est pas utilisé dans JUXMEM. Dans le cas contraire une copie est nécessaire (voir section 7.3.2). Le nom de l'élément dans les couches de communication de JUXMEM est son identifiant.

Par ailleurs, l'utilisation des outils callgrind, greffon pour valgrind [134], et de kcachegrind [225] nous a permis d'identifier les zones de code les plus coûteuses du chemin critique utilisé pour l'envoi ou la réception d'un message JXTA. Chaque zone de code du chemin critique pour l'envoi et la réception d'un message a alors été étudiée, et si besoin réécrite.

Évaluations des optimisations. La figure 10.5 compare les performances en débit du service point-à-point et d'un canal unidirectionnel de JXTA-C 2.2 par rapport à leurs implémentations optimisées. Par ailleurs, cette figure présente également le débit d'une *socket* C en guise de courbe de référence. Nous pouvons constater que l'allure des courbes de débit pour les services optimisés suit celle d'une *socket* C. Les débits du service point-à-point et d'un canal unidirectionnel sont en effet de 110 Mo/s, soit un écart de 6,5 Mo/s par rapport à une *socket* C. Ces valeurs sont obtenues pour des tailles de message de 1 Mo/s et 2 Mo/s pour respectivement le service point-à-point et un canal unidirectionnel. L'augmentation du débit par rapport à l'implémentation de JXTA-C 2.2 non optimisée est de 8 Mo/s et 15 Mo/s, pour respectivement le service point-à-point et un canal unidirectionnel. Cette augmentation est relativement faible.

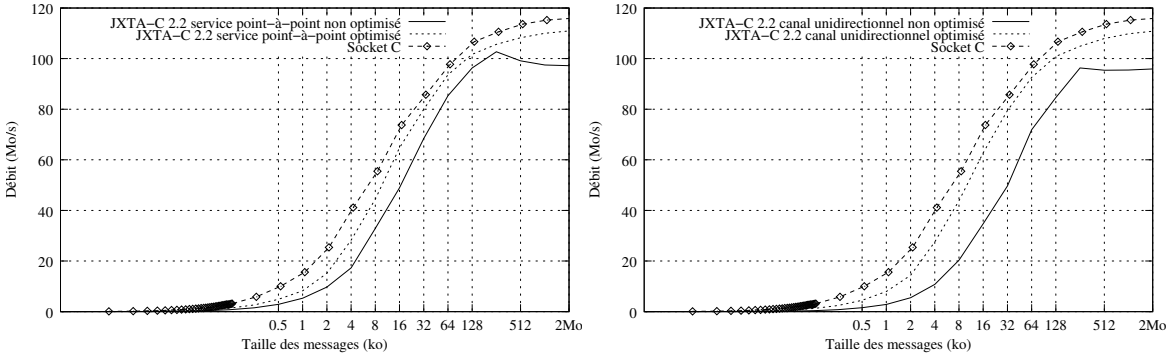

FIG. 10.5 – Débit comparé du service point-à-point (gauche) et d'un canal unidirectionnel (droite) de JXTA-C 2.2 sur un réseau SAN Gigabit Ethernet, avec leurs implémentations optimisées, et en prenant pour courbe de référence le débit d'une *socket* C.

Implémentation	JXTA-C 2.2 non optimisé	JXTA-C 2.2 optimisé
Canal unidirectionnel	294 μs	90 μs
Service point-à-point	149 μs	84 μs
BSD *sockets*	39 μs	

TAB. 10.2 – Performances en termes de latence du service point-à-point et d'un canal unidirectionnel de JXTA-C 2.2, comparées aux performances en latence d'une *socket* C.

En effet, nos modifications[6] visent principalement à optimiser les latences d'envoi et de réception d'un message JXTA. Le tableau 10.2 présente ces performances pour des versions non optimisées et optimisées du service point-à-point et d'un canal unidirectionnel de JXTA-C 2.2. La latence d'un canal unidirectionnel en version optimisée est de 90 μs, soit une amélioration de plus de 200 μs ! Cela est principalement dû à l'optimisation *connexion directe* décrite précédemment. Selon nos mesures, elle représente une amélioration de la latence de 139 μs qui s'explique principalement par l'absence du traitement XML de l'élément *routeur* à chaque envoi/réception d'un message JXTA. Par ailleurs, elle correspond à une réduction de la taille du message de 684 octets, soit une amélioration de l'efficacité du protocole de 71 %. La latence du service point-à-point en version optimisée est de 84 μs, soit un gain de 65 μs. L'optimisation *condensation et mise en cache d'éléments* est la principale explication de cette amélioration de performance. En effet, la mise en cache de l'élément adresse source correspond à une réduction de la taille du message de 69 octets, soit une amélioration de l'efficacité du protocole de 30 %, au niveau du service point-à-point. La condensation de l'élément adresse de destination correspond à gain d'au moins 23 octets. Par ailleurs, les deux services bénéficient de diverses améliorations liées aux réécritures de codes.

La figure 10.6 présente le débit de la couche de communication de JUXMEM-C 0.2 comparé au débit d'un canal unidirectionnel en version optimisée de JXTA-C 2.2, sur lequel elle repose. Par ailleurs et de manière similaire à la figure précédente, le débit d'une *socket* C est également représenté à titre de courbe de référence. La couche de communication de

[6]Excepté l'optimisation mode zéro-copie.

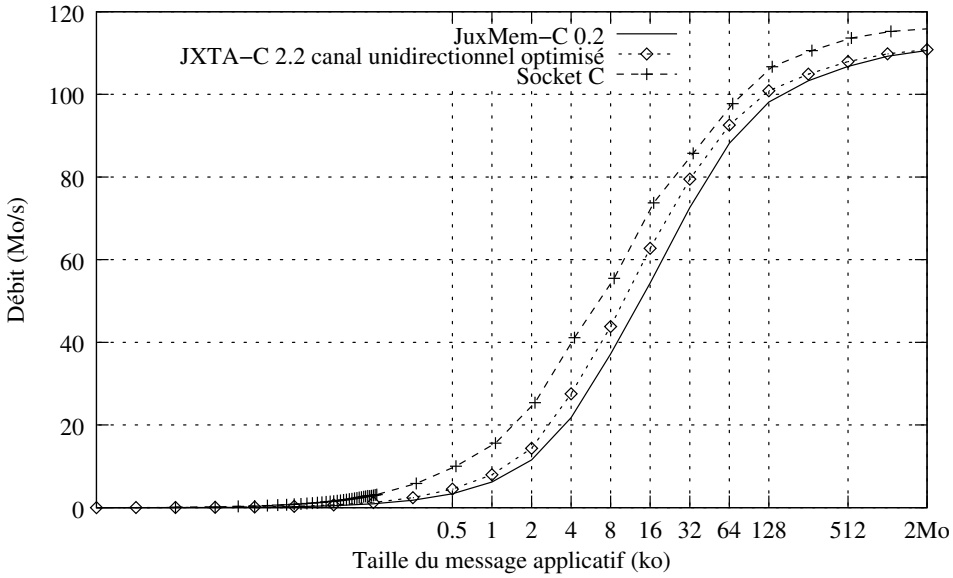

FIG. 10.6 – Débit de la couche de communication de JUXMEM-C 0.2 sur un réseau SAN Gigabit Ethernet, comparé au débit d'un canal unidirectionnel en version optimisée de JXTA-C 2.2 et à une *socket* C.

JUXMEM-C atteint le débit d'un canal unidirectionnel : 110 Mo/s pour une taille de message de 2 Mo/s. La latence est de 127 μs, soit un surcoût de 43 μs par rapport à la latence d'un canal unidirectionnel de JXTA-C. Il s'explique par les deux éléments ajoutés (respectivement lus) pour chaque envoi (respectivement réception) d'un message JXTA. Ces éléments correspondent à l'identifiant du canal unidirectionnel du pair émetteur et à l'identifiant de la fonction de rappel à appeler sur ce pair de destination. Ces éléments ajoutent respectivement 78 et (au minimum) 16 octets à la taille d'un message JXTA, entraînant un surcoût de traitement. Par ailleurs, la création de l'objet JXTA représentant un identifiant est relativement coûteuse. Chaque message JXTA utilisé par JUXMEM utilise son propre espace de nommage rajoutant encore 8 octets à la taille d'un message JXTA. Ainsi, la taille totale d'un message de 1 octet transmis par la couche de communication de est de 303 octets.

Compte tenu des bons résultats de la version initiale de JXTA-C 2.2 sur des réseaux WAN, aucune évaluation de performances de la version optimisée en WAN n'a été effectuée, car aucun changement majeur dans les résultats n'est attendu. Par ailleurs, les performances en débit d'un canal unidirectionnel de JXTA-C étant comparables aux performances de la couche de communication de JUXMEM-C, aucune évaluation de celle-ci en WAN n'a été réalisée non plus.

Discussion. Pour conclure, les optimisations que nous avons réalisées ont permis d'améliorer la latence du service point-à-point et des canaux virtuels de JXTA-C 2.2. Elles constituent un premier pas qui a permis de rendre utilisables ces couches de communication pour des transferts de données efficaces dans JUXMEM-C en SAN. Nos optimisations ont été partiellement intégrées dans JXTA-C 2.3, dernière version sur laquelle JUXMEM-C repose[7]. Par

[7]L'intégration de la totalité des optimisations est en cours (septembre 2006).

ailleurs, les versions postérieures à JXTA-C 2.2 ont également en partie visé l'amélioration des performances de cette implémentation. L'objectif est d'approcher au plus près les performances des *sockets* BSD, tout en gardant les fonctionnalités apportées par JXTA.

10.2.3 Utilisation de la plate-forme haute performance de communication pour grilles PadicoTM

Dans cette section, nous décrivons notre portage de JXTA-C au-dessus de la plate-forme haute performance de communication pour grilles PadicoTM [71]. Par ailleurs, nous présentons également une évaluation de cette réalisation. L'objectif de cette intégration est de pouvoir bénéficier de manière transparente des fonctionnalités avancées fournies par PadicoTM au-dessus des réseaux haute performance des grilles de calcul.

10.2.3.1 Présentation de PadicoTM

PadicoTM, pour *Parallel and distributed Task Manager*, est une plate-forme haute performance de communication pour grilles de calcul. Son principal objectif est d'implémenter le modèle de communications proposé et défini dans [71]. Ce modèle vise à supporter les applications utilisant à la fois les exécutifs qui suivent le paradigme du calcul parallèle et ceux qui suivent celui du calcul réparti. L'objectif de PadicoTM est d'autoriser l'utilisation de n'importe quel exécutif sur n'importe quel type de réseau. Pour ce faire, PadicoTM s'appuie sur des mécanismes de virtualisation et d'abstraction avec *adaptateurs trans-paradigme*. Ainsi, PadicoTM offre l'interface de programmation sur laquelle l'exécutif repose, même si les ressources utilisées pour implémenter cette interface sont différentes. L'objectif est alors pour, par exemple, un intergiciel CORBA de pleinement tirer parti des hautes performances des SAN, sans nécessiter de modifications de son code source (solution habituelle). Par ailleurs, des *adaptateurs alternatifs* permettent de changer, toujours de manière transparente du point de vue de l'exécutif, l'implémentation de l'interface utilisée, en ajoutant par exemple des flux parallèles sur une communication WAN. Enfin, PadicoTM résout des problèmes tels que la cohabitation d'exécutifs, le chargement dynamique de code et l'utilisation efficace du *multi-threading*.

Les intergiciels respectant la spécification JXTA sont des exécutifs du calcul réparti. Deux fonctionnalités de PadicoTM nous intéressent plus particulièrement : 1) l'adaptateur trans-paradigme vers le paradigme du calcul parallèle pour une utilisation transparente des capacités des SAN et 2) les adaptateurs alternatifs qui permettent d'ajouter de manière transparente les flux parallèles et la compression à la volée sur des communications WAN. La première fonctionnalité est celle des *sockets virtuelles*. Elle permet aux exécutifs du réparti d'accéder de manière transparente aux interfaces de programmation des drivers des SAN, tels que Myrinet, Infiniband ou Quadrics par exemple. Les appels aux opérations de l'interface des *sockets* BSD sont alors liés aux opérations de l'interface équivalente fournie par PadicoTM, lors de l'édition de liens dynamiques. Ces dernières reposent directement sur l'interface de programmation du driver de la carte réseau. Une telle fonctionnalité permet de passer outre la couche TCP/IP du noyau et donc d'éviter son surcoût. En conséquence, ces intergiciels peuvent pleinement exploiter les capacités de ces réseaux. Toutefois, l'exploitation de telles capacités n'est possible que si l'intergiciel implémente une couche de communication zéro-copie. À tire d'exemple, ce mécanisme permet sur un réseau Myri-10G,

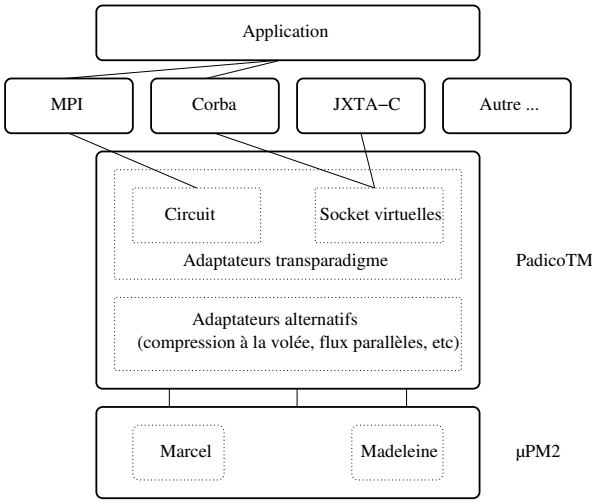

FIG. 10.7 – Empilement des couches logicielles d'une application basée à la fois sur MPI et sur CORBA et qui repose sur PadicoTM.

d'offrir de manière transparente aux *sockets* C une bande passante de 915 Mo/s et une latence de 6 μs[8]. Rappelons que sur un tel réseau, une latence de 2,5 μs peut être atteinte au niveau driver avec une bande passante de 1 150 Mo/s[9]. PadicoTM supporte les intergiciels suivants : deux implémentations de CORBA (OmniORB et MICO), deux implémentations de MPI (MPICH/*Madeleine* et GRIDMPI), la machine virtuelle Kaffe et gSOAP.

La figure 10.7 présente l'empilement des couches logicielles d'une application, basée sur MPI et CORBA, qui repose sur PadicoTM. Elle illustre également les deux types d'adaptateurs présents dans l'architecture interne de PadicoTM. Enfin, PadicoTM repose sur Madeleine [29] et Marcel [132], respectivement une bibliothèque de communication et un ordonnanceur de processus légers. Ces deux bibliothèques forment ce qui est appelé μPM^2 de l'environnement de programmation parallèle par RPC PM^2 [133].

10.2.3.2 Portage de JXTA-C au-dessus de PadicoTM

Porter JXTA-C au-dessus de PadicoTM, via ses *sockets* virtuelles offre donc théoriquement la possibilité d'utiliser de manière transparente les capacités des SAN, en termes de débit et de latence. En outre comme nous l'avons vu dans la section précédente, ce portage permet d'utiliser des adaptateurs alternatifs pour ajouter de manière transparente : des flux parallèles et/ou de la compression à la volée pour tout échange de messages en WAN. Dans cette section, nous décrivons les modifications que nous avons effectuées pour la réalisation du portage de JXTA-C au-dessus de PadicoTM.

Aucune modification dans le code source de JXTA-C n'a été nécessaire. Toutefois, des modifications ont été nécessaires dans la bibliothèque sur laquelle JXTA-C repose. En effet, JXTA-C utilise une bibliothèque d'abstraction des interfaces de programmation fournies

[8] Machines AMD Opteron à 2 GHz avec 2 Go de mémoire vive et tournant sous un Linux 2.6.

[9] Notons toutefois que lors des ces évaluations le support des derniers drivers Myrinet (MX) par PadicoTM était expérimental et en cours d'amélioration (août 2006).

par le système d'exploitation, appelée *Apache Portable Runtime* (APR [196]). Le portage de JXTA-C au-dessus de PadicoTM consiste en réalité à porter APR au-dessus de PadicoTM. L'objectif d'APR est de fournir aux développeurs une interface de programmation leur assurant un comportement prévisible, si ce n'est identique, et indépendant de la plate-forme sur laquelle s'exécutent leurs applications. Ainsi, les développeurs n'ont pas à se préoccuper du codage des cas particuliers pour pallier ou utiliser des carences ou des fonctionnalités spécifiques à une plate-forme d'exécution. L'interface de programmation mise à disposition couvre le chargement de bibliothèque dynamique, les opérations sur les fichiers et sur la mémoire, la programmation réseau, les processus lourds et légers, diverses structures de données, etc. APR est implémenté dans le langage C et peut s'exécuter sur Windows, Netware, Mac OS X, OS/2 et les variantes existantes d'Unix. JXTA-C utilise principalement d'APR les processus légers, les *sockets* BSD et diverses structures de données.

Comme nous l'avons décrit dans la section précédente, PadicoTM est basé sur la bibliothèque de processus léger Marcel. Ainsi, l'utilisation de la bibliothèque *native POSIX Thread* (NPTL), choisie par défaut par APR sous les variantes d'Unix, n'est pas possible : une seule bibliothèque pouvant être utilisée. Cependant, PadicoTM fournit les commandes nécessaires pour « marceliser » les primitives *pthread* utilisées par APR. Le prototype des fonctions commençant par pthread est remplacé par marcel. Malheureusement, l'utilisation de ces commandes n'est pas suffisante. En effet, l'utilisation réalisée par APR de la spécification des processus légers POSIX est exhaustive. Or, l'implémentation fournie par la bibliothèque Marcel de cette spécification n'est pas complète, tout du moins via l'interface marcel de Marcel. Les verrous récursifs ne sont notamment pas supportés dans cette interface, mais uniquement dans l'interface pmarcel de Marcel. Une modification dans le code d'APR a donc été nécessaire afin de distinguer le type de verrou utilisé. Par ailleurs, la gestion des signaux n'est pas entièrement implémentée dans Marcel : certains appels ne sont pas disponibles. En conséquence, les fonctions APR de gestion des signaux retournent une constante indiquant que ceux-ci ne sont implémentés. Toutefois, l'intergiciel JXTA-C n'utilise pas de signaux pour sa mise en œuvre. Enfin, les prototypes des fonctions pour l'acquisition et le relâchement d'un sémaphore entre les bibliothèques respectant la spécification des processus légers POSIX et Marcel sont différents, nécessitant également une modification du code d'APR.

La version 1.2.1 d'APR a été portée sur PadicoTM version 0.3 beta 2, permettant à la version 2.3 de JXTA-C en mode optimisé d'être testée. Compte tenu de l'absence de modifications au sein de JXTA-C, nous pensons que les versions supérieures de cet intergiciel peuvent être immédiatement supportées par PadicoTM. D'autre part, les minimes modifications apportées à APR nous permettent de croire que les versions 1.2.x peuvent être portées très rapidement. L'implémentation Java de JXTA n'a pas été portée au-dessus de PadicoTM. En effet, PadicoTM ne supporte actuellement que la machine virtuelle Kaffe. Or, celle-ci n'implémente pas la spécification 1.4 requise par les versions de JXTA-J2SE inférieures à 2.3.7. En outre, à partir de la version 2.3.7, JXTA-J2SE nécessite une machine virtuelle respectant la spécification 5.0 de Java.

10.2.3.3 Évaluations

Dans cette section nous présentons les évaluations que nous avons faites de notre portage de JXTA-C et JUXMEM-C au-dessus de PadicoTM.

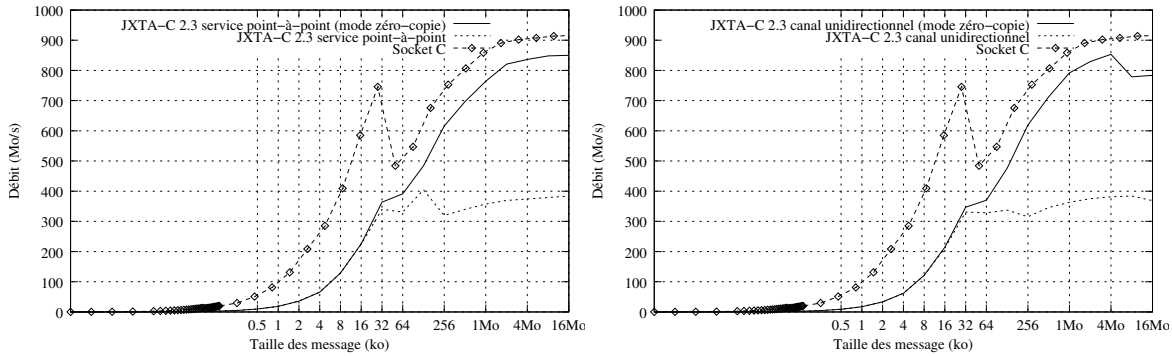

FIG. 10.8 – Débits comparés du service point-à-point (gauche) et d'un canal unidirectionnel (droite) de JXTA-C 2.3 en version optimisée au-dessus de PadicoTM sur un réseau Myri-10G, avec ou sans l'option zéro-copie activée, et en prenant pour courbe de référence le débit d'une *socket* C.

Implémentation	JXTA-C 2.3 (avec PadicoTM)
Service point-à-point	47 μs
Canal unidirectionnel	52 μs
BSD *sockets*	6 μs

TAB. 10.3 – Performances en termes de latence du service point-à-point et d'un canal unidirectionnel de JXTA-C en version optimisée avec l'utilisation de PadicoTM et sur un réseau Myri-10G, comparées aux performances en latence d'une *socket* C.

Évaluation des couches de communication de JXTA-C. La figure 10.8 compare les performances en termes de débit du service point-à-point et d'un canal unidirectionnel de JXTA-C 2.3 en version optimisée au-dessus de PadicoTM, avec ou sans l'option zéro-copie activée. Par ailleurs, cette figure présente également le débit d'une *socket* C au-dessus de PadicoTM en guise de courbe de référence. Nous pouvons constater que l'allure des courbes de débit lorsque l'option zéro-copiée est activée suit celle d'une *socket* C. En effet, les débits du service point-à-point et d'un canal unidirectionnel atteignent la valeur de 850 Mo/s, soit un écart de 75 Mo/s par rapport à une *socket* C. En revanche, les débits du service point-à-point et d'un canal unidirectionnel lorsque le mode zéro-copie n'est pas activé stagnent à une valeur de 380 Mo/s. Ainsi, l'utilisation conjointe de notre optimisation zéro-copie pour JXTA-C et de PadicoTM permet de pleinement tirer parti des performances des SAN en termes de débit. Cela constitue une amélioration du débit des couches de communication de JXTA par un facteur de 2,2. L'impact sur les débits de notre optimisation zéro-copie est donc significative.

En termes de latence, l'utilisation de PadicoTM permet comme attendu d'éviter le surcoût du passage par la couche TCP/IP du noyau. En effet, les gains sont de 37 μs et de 38 μs pour respectivement le service point-à-point et un canal unidirectionnel de JXTA-C 2.3 au-dessus de PadicoTM. Ces valeurs correspondent bien à la latence d'une *socket* C ne s'exécutant pas au-dessus de PadicoTM (pour rappel de 39 μs, voir tableau 10.2). Le tableau 10.3 résume ces performances.

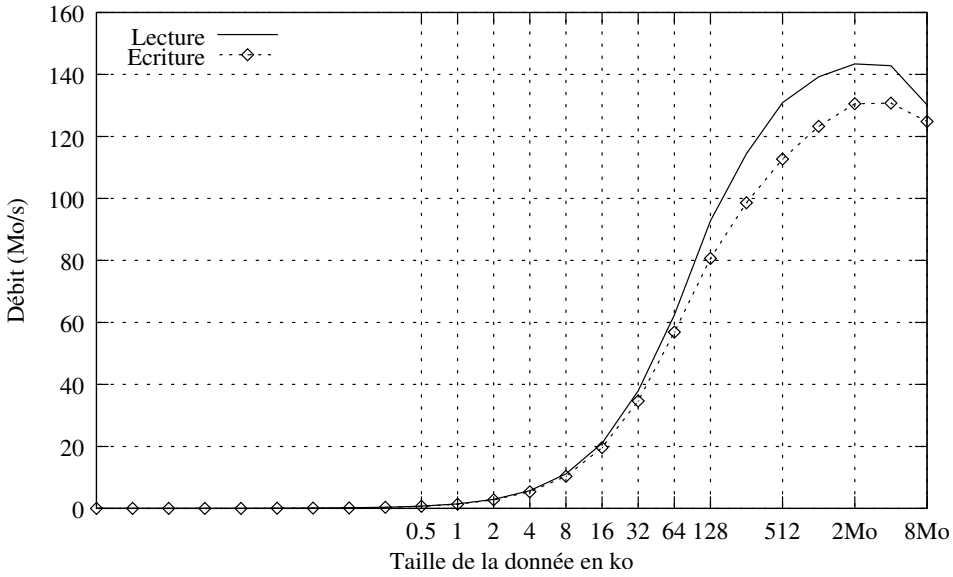

FIG. 10.9 – Débits des opérations de lecture et d'écriture de JUXMEM-C 0.3 sur un réseau Myri-10G au-dessus de PadicoTM.

Évaluation de l'impact sur les performances de JUXMEM. L'évaluation que nous avons réalisée des performances de la couche de communication de JUXMEM-C 0.3, sur un réseau Myri-10G au-dessus de PadicoTM, montre des performances en termes de débit similaires à celles d'un un canal unidirectionnel de JXTA-C (850 Mo/s). La latence est améliorée de 55 μs et atteint une valeur de 72 μs. Cette diminution est supérieure à celle attendue et s'explique par des optimisations mineures réalisées dans la couche de communication de JUXMEM-C entre la version 0.2 et 0.3. Ces optimisations consistent en une réduction de la taille totale d'un message de 1 octet applicatif.

La figure 10.9 présente les débits des opérations de lecture et d'écriture de JUXMEM-C 0.3 sur un réseau Myri-10G et au-dessus de PadicoTM. Le débit maximal d'une lecture est de 143 Mo/s, alors que le débit maximal d'une écriture est de 130 Mo/s. Il sont atteints pour une taille de données de 2 Mo dans les deux cas. Par rapport aux résultats du même test sur un réseau Gigabit Ethernet (voir section 7.5), les gains en termes de débit pour une lecture et une écriture sont respectivement de 51 Mo/s et 38 Mo/s (soit des gains respectifs de 36 % et 30 %). Ces gains sont relativement faibles et s'expliquent principalement par le nombre de messages réseaux nécessaires pour une opération de lecture ou d'écriture : 4 messages dans les deux cas (voir section 7.5 pour plus d'explications). Toutefois, une évaluation des zones de code les plus coûteuses grâce aux outils callgrind et kcachegrind est nécessaire pour confirmer cette hypothèse ainsi qu'améliorer les latences de traitement de la couches, encore expérimentale, de gestion de tolérance aux fautes dans JUXMEM-C 0.3. Par ailleurs, la latence d'une lecture est de 660 μs. La latence d'une écriture est de 710 μs. Cette absence de gain, pour l'opération de lecture, du fait de la non traversée de la couche TCP/IP du noyau n'est pas expliquée à ce jour. La dégradation de la latence pour l'opération d'écriture n'est également pas expliquée.

10.3 Discussion et conclusion

Dans ce chapitre, nous avons tout d'abord présenté le fonctionnement interne des couches de communication de JXTA, telles qu'elles sont implémentées dans JXTA-C et JXTA-J2SE. Avant de détailler les optimisations que nous avons réalisées dans ces couches, nous avons introduit notre motivation ainsi que notre méthodologie de test. Notre démarche s'inscrit dans un contexte de convergence des systèmes P2P et des intergiciels pour les grilles de calcul. Elle fait naître, dans un premier temps, le besoin de tester l'adéquation des bibliothèques P2P pour le développement d'intergiciels pour les grilles de calcul. Puis, dans un deuxième temps, apparaît la nécessité d'adapter ces bibliothèques P2P aux grilles de calcul, notamment afin de tirer parti des capacités offertes par les réseaux haute performance disponibles dans ces infrastructures. Pour ce faire, nous avons détaillé les optimisations que nous avons réalisées ainsi que leurs impacts sur les performances des couches de communication de JXTA-C, mais également de JUXMEM-C. Enfin, dans une dernière étape pour pleinement et de manière transparente tirer parti des capacités des réseaux haute performance, nous avons montré comment obtenir des communications zéro-copie grâce au portage de JXTA-C au-dessus de la plate-forme haute performance de communication pour grilles PadicoTM. Afin de valider ce portage, des évaluations sur divers types de réseaux ont été menées sur la grille expérimentale Grid'5000, infrastructure présentée au chapitre 2.

L'évaluation que nous avons menée nous a permis de constater les mauvaises performances initiales des couches de communication de JXTA, notamment en termes de latence. Nous avons montré comment nous y avons remédié par diverses optimisations. Nous avons ainsi démontré l'adéquation des bibliothèques P2P dans le contexte des grilles de calcul, en particulier pour le transfert de données. En effet, les débits obtenus répondent aux exigences des performances des applications de calcul sur grilles. En particulier, l'utilisation conjointe : 1) de notre possibilité d'utiliser les couches de communications de JXTA-C en mode zéro-copie et 2) de PadicoTM, permet de pleinement tirer parti des performances des SAN en termes de débit. En outre, ces optimisations nous ont permis d'améliorer les performances de la couche de communication de JUXMEM, et en conséquence, d'améliorer les performances des lectures et des écritures au sein de JUXMEM.

Toutefois, le scénario de déploiement que nous considérons pour notre test bidirectionnel de bande passante suppose des communications directes entre toutes les machines. Ceci n'est toutefois pas toujours possible. En effet, sur certaines grilles de calcul les communications entre les machines d'une grappe ne peuvent avoir lieu que par l'intermédiaire des machines dits *frontaux* de chaque grappe. Dans ce cas, d'autres évaluations sont nécessaires.

Concernant l'implémentation du portage de JXTA au-dessus de PadicoTM, les modifications qu'il faut apporter à APR pour l'utilisation de Marcel ne seront plus nécessaires dans le futur. En effet, des travaux sont actuellement en cours dans la bibliothèque de processus léger Marcel pour un meilleur support de l'interface pthread. Un meilleur support des signaux est également en cours d'implémentation, ce qui permet d'envisager à terme le portage de JXTA-J2SE au-dessus de PadicoTM. Enfin, le portage que nous avons réalisé d'APR permet également d'envisager le support dans PadicoTM d'autres intergiciels se basant sur APR. Nous pouvons notamment citer le serveur Web majoritairement utilisé aujourd'hui : Apache. Ce travail ouvre donc des perspectives pour, par exemple, l'équilibrage de charge entre différents serveurs Web et cela de manière transparente sur les SAN.

Chapitre 11

Déploiement d'applications JXTA sur une infrastructure de type grille

Sommaire

Le dernier chapitre de cette partie présente notre contribution à l'utilisation d'applications basées sur JXTA et s'exécutant sur une infrastructure de type grille. À cet effet, nous avons défini un langage qui permet de décrire le réseau de pairs à déployer à partir d'une application basée sur JXTA. Ensuite, afin de valider notre proposition, nous avons évalué le passage à l'échelle d'un protocole de la spécification JXTA. Toutefois, la première motivation de ces travaux a été de permettre le déploiement de JUXMEM à grande échelle, notamment sur la grille expérimentale Grid'5000 [51]. Ainsi, les expérimentations présentées dans le chapitre 8 ont pu être menées grâce à la contribution qui fait l'objet de ce chapitre.

Dans un premier temps, nous présentons le contexte de ce travail : la convergence entre grilles de calcul et environnements pair-à-pair (P2P). Puis, nous introduisons les étapes qui nous ont permis d'arriver à notre contribution actuelle : la proposition d'un langage de description des réseaux virtuels JXTA. Ensuite, nous décrivons la mise en œuvre de ce langage au sein de l'outil de déploiement générique ADAGE, à travers un exemple d'utilisation. En

guise de validation et comme nous l'avons indiqué précédemment, nous présentons les résultats de l'évaluation du passage à l'échelle d'un protocole de la spécification JXTA. Enfin, nous terminons par une discussion sur notre proposition.

11.1 Utilisation d'une approche pair-à-pair sur les grilles de calcul

L'utilisation d'une approche P2P pour la programmation d'intergiciels pour les grilles de calcul, énoncée dans la section 2.5 et rappelée dans la section 10.2.1, constitue notre contexte de travail pour cette contribution. Ainsi, comme nous l'avons vu dans ces sections, l'idée d'utiliser des mécanismes P2P pour bâtir les intergiciels pour les grilles de calcul est naturelle. En effet, les propriétés du modèle P2P, telles que le passage à l'échelle et la tolérance à la volatilité manquent aux intergiciels pour grilles de calcul. Cette utilisation peut être réalisée selon deux manières différentes.

- La première possibilité consiste à intégrer dans les couches basses des systèmes P2P les intergiciels pour grilles de calcul. Par exemple, ils peuvent utiliser les couches de communication des intergiciels pour grilles de calcul pour exposer leurs services P2P de plus haut niveau [171]. Toutefois, de telles couches de communication ne tiennent généralement pas compte de la volatilité des entités du système, qu'elles supposent stables en général. Or, l'intégration de cette volatilité est nécessaire pour le passage à l'échelle et la flexibilité de l'intergiciel.
- La seconde possibilité est d'utiliser les bibliothèques P2P comme fondation pour bâtir des services pour grilles de calcul de plus haut niveau [19]. Comme nous l'avons illustré dans le chapitre précédent à travers les couches de communication de JXTA, une telle utilisation peut nécessiter une adaptation des bibliothèques P2P à l'environnement d'exécution. C'est cette approche que nous prônons pour l'utilisation des mécanismes P2P sur les grilles de calcul.

Quelle que soit l'approche choisie, le problème du déploiement d'applications utilisant des bibliothèques P2P se pose, et cela à deux titres.

1. Contrairement à certains modèles de programmation, tels que les modèles composant par exemple, le modèle P2P ne définit pas un modèle d'empaquetage et de déploiement des services offerts par un ensemble de pairs. Cela s'explique par les origines de la popularité des systèmes P2P : les systèmes de partage de fichiers. Dans de telles applications, c'est l'utilisateur qui va lancer le système P2P généralement spécifique et qui offre comme unique service celui du partage de fichiers. Ce n'est pas le cas dans le contexte des grilles de calcul où les utilisateurs souhaitent déléguer le déploiement et l'exécution d'applications aux intergiciels gérant ce type d'infrastructures. Ceux-ci doivent donc savoir déployer une application partiellement bâtie sur un modèle P2P et qui peut éventuellement interagir avec plusieurs services P2P. Une telle connaissance est nécessaire pour que les processus ainsi déployés établissent leurs connexions initiales : charge à la bibliothèque P2P utilisée de faire évoluer, par la suite, cette topologie de départ.
2. De par les ressources qu'elles mettent à disposition, les grilles de calcul offrent un excellent cadre pour la mise au point et l'évaluation de mécanismes P2P. De telles expérimentations doivent être réalistes et nécessitent donc l'utilisation d'un grand nombre

de machines, afin d'émuler jusqu'à plusieurs centaines de milliers de pairs. Le déploiement d'un aussi grand nombre de processus sur les milliers de machines qu'une grille peut agréger reste un défi.

Dans les deux cas, l'utilisation d'un langage adéquat pour décrire le réseau P2P à déployer est nécessaire. Compte tenu de l'échelle visée, potentiellement grande, il doit permettre une description condensée mais brève des pairs formant le réseau virtuel. À notre connaissance, de tels langages n'existent pas et n'ont pas été testés à l'échelle d'une grille de calcul.

11.2 Déploiement d'applications basées sur JXTA

Dans cette section, nous introduisons notre proposition pour la description d'applications basées sur JXTA, ainsi que sa mise en œuvre dans un outil de déploiement. De manière analogue au chapitre précédent, le choix d'étudier JXTA comme modèle P2P est guidé par son utilisation dans la conception et l'implémentation de JUXMEM. Notre objectif est d'abord de permettre le déploiement de JUXMEM à grande échelle pour son évaluation, mais également de fournir une solution générique utilisable par toute application basée sur JXTA.

11.2.1 Une première approche : JDF

Lors de nos premières évaluations de JUXMEM sur quelques machines, Sun Microsystems développait un outil pour automatiser le test d'applications basées sur JXTA. Nous avons donc tout d'abord évalué cet outil, appelé JDF [217] (pour JXTA *Distributed Framework*), en vue d'une éventuelle utilisation pour effectuer nos expérimentations à plus grande échelle. Le projet JDF a servi d'incubateur au développement d'une bibliothèque de classes spécialisées pour la configuration d'un pair JXTA-J2SE, appelée ext:config [219]. Cette bibliothèque implémentée en Java permet de décrire la configuration d'un pair soit sous la forme d'un fichier XML, appelé profile, soit par programmation. Toutefois, elle vise uniquement la *configuration* d'un pair JXTA-J2SE (rendez-vous, standard ou relais), c'est-à-dire qu'elle ne prend pas en compte la problématique du *déploiement* d'un ensemble de pairs.

JDF est un environnement qui permet de définir uniquement des configurations de réseaux virtuels JXTA-J2SE[1] à tester, puis de les déployer sur un ensemble de ressources distribuées. Cet outil est constitué d'un ensemble de scripts shell qui effectuent une série d'étapes élémentaires : installation des fichiers, initialisation du réseau JXTA-J2SE, lancement du test, rapatriement des résultats produits, analyse de ces derniers et nettoyage des machines utilisées. JDF nécessite qu'un tel test distribué soit constitué des éléments décrits ci-dessous.

- Un fichier de description du réseau virtuel de pairs JXTA à déployer. Ce fichier décrit, par exemple, les types des pairs utilisés pour le test, leurs interconnexions, etc. La notion de *profil* permet de factoriser les informations communes à un ensemble de pairs s'exécutant sur une même machine.
- Un ensemble de classes Java contenant le code à exécuter pour chaque type d'entité impliqué dans le test. Ces classes doivent étendre un canevas fourni par JDF qui permet de démarrer et arrêter JXTA, ainsi que de sauvegarder les résultats.

[1] Et non JXTA-C par exemple.

– Un fichier contenant la liste des machines à utiliser pour le déploiement des pairs.
– Un fichier contenant la liste des bibliothèques Java (fichiers .jar) à déployer.

JDF nous a permis de réaliser les premiers tests de notre prototype JUXMEM-J2SE, au sein d'une grappe de machines mais également en utilisant plusieurs grappes. Cependant, cet outil a très rapidement montré ses limites. En effet, le langage de description utilisé ne permettait pas à un ensemble de pairs distribués sur plusieurs machines de partager un même profil. Par ailleurs, JDF n'était pas en mesure d'interagir avec les gestionnaires de ressources, tels que PBS et Globus, pour la soumission d'un test. Nous avons entrepris de résoudre ces points, notamment pour le bon déroulement de nos expérimentations liées à JUXMEM. Par exemple, nous avons introduit dans la définition d'un profil JDF la notion d'instance. Elle permet le déploiement, sur plusieurs machines, de pairs issus du même profil.

Toutefois, malgré nos optimisations, les performances de déploiement de JDF n'ont pas atteint un niveau suffisant. Son passage à l'échelle s'est révélé en pratique peu concluant. De plus, toute intégration de modifications est pénible et longue, plus particulièrement lorsqu'il s'agit du langage de description de JDF. Outre les problèmes de mise en œuvre, JDF souffre de choix conceptuels non judicieux dans son langage de description. En effet, celui-ci mélange la définition des profils et leurs utilisations, rendant la définition de tests confuse et empêchant une réutilisation à divers endroits du même profil. Chaque ensemble de pairs standards connecté à un pair rendez-vous nécessite par ailleurs un propre profil, même si ce dernier est identique. Enfin, JDF est initialement conçu pour le format des fichiers de configuration de JXTA-J2SE. Or, ce format diffère légèrement de celui utilisé par JXTA-C, nécessitant cependant de lourdes modifications dans le code de JDF ainsi que dans le langage de description de JDF. Au regard de l'ensemble de ces problèmes, nous avons été contraints d'abandonner l'utilisation de JDF.

11.2.2 Proposition d'un langage de description

Forts de cette première expérience, nous avons redéfini un langage pour la description de réseaux virtuels JXTA, qui ne présente pas ces déficiences. Dans cette section, nous présentons ce langage que nous appelons JDL, pour JXTA *Description Langage*.

Chaque fichier utilisant JDL pour décrire un test est composé de deux parties. La première partie correspond à la définition de l'ensemble des profils (profiles) qui seront utilisés dans le test. Un profil de pairs (profile) est défini par le type de pair qu'il représente, ses point d'accès, le type de son exécutable et le chemin vers cet exécutable. Les types d'exécutables gérés sont des binaires écrits dans le langage C/C++ ainsi que des classes Java. Chaque profil de pairs est désigné de manière unique par un identifiant. La deuxième partie d'une description au format JDL comporte l'utilisation qui est faite de ces profils : l'instanciation en un ensemble de pairs (overlay). Un ensemble de pairs emprunte les caractéristiques du profil de pairs qu'il référence. Il dispose en outre d'un identifiant ainsi que d'une cardinalité correspondant au nombre de pairs de l'ensemble. Actuellement, les pairs supportés sont les pairs rendez-vous (rdvs) et les pairs standards(edges). L'interconnexion initiale dans le réseau virtuel à déployer est également précisée. Ainsi, la vue locale des pairs rendez-vous est spécifiée lorsqu'il s'agit d'un pair rendez-vous. Lorsqu'il s'agit d'un pair standard, le pair rendez-vous auquel il doit se rattacher est précisé.

La figure 11.1 représente un exemple de description d'un réseau virtuel JXTA basé sur le langage JDL.

Listing 11.1 – Exemple de description d'un réseau virtuel basé sur le langage JDL.

```
1  <JXTA_application>
2  <profiles>
3     <profile name="p_rdv">
4          <services rdv="true" relay="false" />
5          <networks tcp="true" multicast="false" http="false"/>
6          <behavior binding="c" filename="manager" args="1800"/>
7     </profile>
8     <profile name="p_edge">
9          <services rdv="false" relay="false" />
10         <networks tcp="true" multicast="false" http="false"/>
11         <behavior binding="c" filename="provider" args="10000␣1800"/>
12    </profile>
13 </profiles>
14
15 <overlay>
16    <rdvs>
17      <rdv id="r1" profile_name="p_rdv" cardinality="1">
18      </rdv>
19      <rdv id="r2" profile_name="p_rdv" cardinality="1">
20         <rpv>r1</rpv>
21      </rdv>
22      <rdv id="r3" profile_name="p_rdv" cardinality="1">
23         <rpv>r2</rpv>
24      </rdv>
25    </rdvs>
26    <edges>
27      <edge id="e1" profile_name="p_edge" cardinality="1" rdv="r1"/>
28      <edge id="e1" profile_name="p_edge" cardinality="2" rdv="r2"/>
29      <edge id="e1" profile_name="p_edge" cardinality="3" rdv="r3"/>
30    </edges>
31 </overlay>
32 </JXTA_application>
```

Le test est constitué de 3 pairs rendez-vous $R1$, $R2$ et $R3$ (lignes 17, 19 et 22), et de leur ensemble de pairs standards associés $E1$, $E2$ et $E3$ (lignes 27 à 29). Ces ensembles sont respectivement d'une cardinalité de 1, 2 et 3 pairs standards. Toutefois, uniquement deux profils sont définis (lignes 3 et 8). En effet, les pairs rendez-vous et les pairs standards utilisent chacun une même classe de test. La topologie initiale est un chemin entre les pairs rendez-vous. Cet exemple se base sur des binaires écrits en C/C++ et donc liés à JXTA-C. Par ailleurs, ce test est d'une taille relativement réduite et peut s'exécuter sur une grappe. Cependant, il peut être facilement étendu afin d'atteindre l'échelle d'une grille en instanciant d'autres pairs rendez-vous et en modifiant la valeur de l'attribut cardinality pour chaque ensemble de pairs standards. L'annexe A.4 présente la définition du langage de description JDL d'applications basées sur JXTA.

L'exemple précédent décrit un réseau virtuel JUXMEM, les exécutables référencés par l'attribut filename correspondant à des binaires JUXMEM-C. Cependant, nous tenons à faire remarquer que le langage JDL proposé n'est pas spécifique à notre service de partage de données. Il s'applique aux applications ou services basés sur la spécification JXTA et implémentés sur JXTA-J2SE, JXTA-C, ou à un mélange des deux.

11.2.3 Mise en œuvre au sein de l'outil de déploiement ADAGE

Afin d'évaluer notre proposition de langage, nous l'avons mise en œuvre au sein de l'outil de déploiement générique ADAGE [114] (pour *Automatic Deployment of Applications in a Grid Environment*). L'objectif de cet outil est d'automatiser le déploiement sur les grilles d'applications quel que soit le type de paradigme sur lequel elles s'appuient : distribué, parallèle, voire un mélange des deux (composants parallèles par exemple). Cet aspect générique est le principal avantage d'ADAGE par rapport aux autres outils de déploiement, qui visent généralement un type d'application en particulier. Pour obtenir ce caractères générique, chaque type d'application est décrit dans son propre formalisme. Les principales étapes pour le déploiement d'applications sont alors les suivantes : 1) conversion de la description de l'application en un modèle de description générique, compréhensible par ADAGE et appelé GADe (pour *Generic Application Description*), 2) planification, c'est-à-dire choix du placement des processus sur les ressources dont la description a été donnée en entrée, 3) exécution du plan de déploiement ainsi construit à l'étape précédente et 4) configuration de l'application. L'outil ADAGE permet par ailleurs à l'utilisateur de conserver un certain contrôle sur le processus de déploiement, tel que le choix du planificateur par exemple. C'est le rôle des paramètres de contrôle d'ADAGE.

ADAGE est en mesure de déployer des applications basées sur CCM, MPICH1-P4, MPICH-G2, GRIDCCM et, suite à nos efforts, JXTA. Chaque type d'application supportée dispose d'une description spécifique qui contient toutes les informations utiles pour leur déploiement automatique. Par exemple, la figure 11.1 de la section précédente correspond à une description spécifique d'applications basées sur JXTA. GADe lui vise à s'abstraire de tout concept spécifique aux applications. Il se résume à décrire les applications sous la forme d'ensembles de processus (éventuellement de processus légers également) distribués sur une ou plusieurs machines, et éventuellement connectés s'ils communiquent entre eux.

L'ajout du support de JXTA au sein d'ADAGE consiste en l'écriture d'un greffon (en anglais *plugin*) capable de convertir une description JDL vers GADe et de configurer l'ensemble des processus déployés. Notre mise en œuvre de ces deux rôles est décrite ci-dessous.

Convertisseur JDL vers GADe. L'objectif est de convertir les informations contenues dans une description au format JDL en une description GADe. Le code correspondant est relativement simple à écrire puisque chaque pair JXTA s'exécute dans un processus distinct. Il consiste principalement en l'expansion du champ cardinality en processus, tel qu'il est défini dans le format utilisé par GADe. Par ailleurs, les paramètres de contrôle permettent éventuellement de préciser la distribution de ces processus sur un ensemble de ressources, en autorisant plusieurs pairs à s'exécuter sur une même machine. 800 lignes de code dans le langage C++ ont été nécessaires pour cette mise en œuvre.

Configuration. L'objectif est de configurer chaque processus déployé, c'est-à-dire de générer le fichier de configuration du pair associé. Par exemple, l'adresse du pair rendez-vous doit être spécifiée pour un pair standard, les identifiants de chaque pair doivent être uniques, etc. Outre la description spécifique de l'application, cette phase s'appuie sur la description des ressources pour déterminer l'interconnexion TCP/IP initiale des pairs par exemple. Les paramètres de contrôle peuvent intervenir également lors de cette phase pour, par exemple, le placement des pairs sur telle ou telle grappe de machines, si besoin. Le code correspondant représente 600 lignes de code C++. Il permet

la configuration de pairs JXTA-C et JXTA-J2SE à partir de fichiers de configuration prédéfinis (en anglais *template*).

Au total, le greffon JXTA pour ADAGE représente 1400 lignes de code C++. Il a permis de valider la généricité de cet outil de déploiement et de mesurer la simplicité d'écriture d'un greffon pour un type d'intergiciel donné.

11.3 Évaluation du passage à l'échelle de JXTA

Dans cette section, nous présentons la validation de notre proposition de langage JDL. Pour ce faire, nous avons évalué le passage à l'échelle d'un protocole spécifié par JXTA : le protocole de gestion de vue.

11.3.1 Protocole de gestion de vue

Dans les implémentations de référence de JXTA, la structure d'un réseau virtuel est organisée par les pairs rendez-vous. Pour rappel (voir section 6.2.1), dans le contexte d'un groupe, chaque pair rendez-vous dispose d'une vue des autres pairs rendez-vous membres de ce groupe. Supposons que le nombre de pair rendez-vous soit de n, alors notre système P2P $\mathbf{S}$ est formé de l'ensemble des pairs rendez-vous (notés R_i) :

$$\mathbf{S} = \{R_i, i = 1..n\} \tag{11.1}$$

Le système $\mathbf{S}$ définit une *vue globale* constitué de l'ensemble de pairs rendez-vous. Notons g la taille de vue globale. Par ailleurs, notons PV_{R_i} la *vue locale* du pair rendez-vous R_i et l_i la taille de PV_{R_i}. Enfin, notons $\mathbf{T}$ l'ensemble des valeurs prises par le temps qui s'écoule pour le système $\mathbf{S}$. L'objectif du protocole de gestion de vue est alors de construire et de maintenir cohérentes les vues locales des pairs rendez-vous. Nous pouvons traduire cette propriété par l'expression suivante :

$$\exists\, t_1 \in \mathbf{T}, \forall\, t_2 \in \mathbf{T}, t_2 > t_1, \forall\, R_i \in \mathbf{S} : l_i = g \tag{11.2}$$

Cette propriété de cohérence est nécessaire pour un routage optimal des requêtes de découverte (voir annexe A.2.2). Pour obtenir cette convergence des vues locales, les pairs rendez-vous testent périodiquement les autres membres de leur vue.

L'algorithme 1 montre le code utilisé à cet effet dans JXTA-C[2]. Avant de dérouler cet algorithme, nous introduisons les différents paramètres qui le contrôlent. L'intervalle de temps entre les exécutions de l'algorithme est spécifié par la constante PEERVIEW_INTERVAL (valeur par défaut de 30 secondes). La constante PVE_EXPIRATION contrôle le temps maximum pendant lequel une entrée est stockée dans la vue locale d'un pair rendez-vous (valeur par défaut de 20 minutes). Les variables upper_rdv et lower_rdv correspondent respectivement au pair rendez-vous supérieur et au pair rendez-vous inférieur du pair rendez-vous local[3], dans sa vue locale. La constante HAPPY_SIZE indique la taille minimale que doit avoir la vue locale pour que l'algorithme change de comportement (valeur par défaut de 4).

[2]Notons que cet algorithme est légèrement différent de celui utilisé dans JXTA-J2SE, que nous ne détaillons pas ici.

[3]En suivant l'ordre lexicographique sur les identifiants des pairs.

Algorithme 1 : Pseudo-code des principales étapes de l'algorithme périodiquement utilisé pour la convergence de la vue locale des pairs rendez-vous.

```
1  répéter
2      attendre pendant PEERVIEW_INTERVAL;
3      détruire les entrées de la vue locale dont le temps > PVE_EXPIRATION;
4      n = taille de la vue locale;
5      pour chaque rdv ∈ {upper_rdv, lower_rdv} faire
6          si n < HAPPY_SIZE alors
7              tester le pair rdv;
8          sinon
9              si rand() % 3 == 0 alors
10                 mettre à jour notre entrée dans la vue locale du pair rdv;
11             sinon
12                 tester le pair rdv;
13     si n < HAPPY_SIZE alors
14         tester les pairs rendez-vous graines (en anglais seeds rendezvous);
15 jusqu'à service rendez-vous n'est plus actif ;
```

Pour atteindre son objectif, l'algorithme *teste* les différentes entrées de la vue locale d'un pair rendez-vous. Un pair rendez-vous R_1 teste un pair rendez-vous R_2 en lui envoyant son annonce de pair rendez-vous R_1. En réponse à un test, un pair rendez-vous retourne sa propre annonce et, dans un message différent, l'annonce d'un autre pair rendez-vous pris aléatoirement dans sa vue. Ce second message est connu sous le nom de *réponse de référence* (en anglais *referral response*). Il permet donc aux pairs de découvrir d'autres pairs rendez-vous et ainsi d'espérer dépasser le seuil HAPPY_SIZE pour la taille de leur vue locale. Toutefois, avant d'ajouter cette référence de pair rendez-vous dans sa vue locale, le pair rendez-vous doit également le tester pour s'assurer qu'il est toujours vivant.

L'algorithme se découpe donc en deux cas, selon la taille n de la vue locale.

1. $n < HAPPY\_SIZE$. Dans ce cas, s'il existe un pair upper_rdv et/ou un pair lower_rdv ils sont testés (ligne 7). Par ailleurs, le pair rendez-vous local teste également les pairs rendez-vous avec lesquels il a été initialement configuré lors de son démarrage (ligne 14). Ceci correspond au mécanisme d'amorçage de l'algorithme. Ces pairs rendez-vous sont appelés des pairs rendez-vous graines (en anglais *seeds rendezvous*).
2. $n \geq HAPPY\_SIZE$. L'algorithme teste alors uniquement le pair upper_rdv et le pair lower_rdv (lignes 9 à 12). Trois sous-cas d'interaction avec ces pairs se produisent.
   (a) Dans $\frac{4}{9}$ des cas les deux pairs sont testés.
   (b) Dans $\frac{4}{9}$ des cas, un des deux pairs est testé et l'entrée du pair rendez-vous dans la vue locale de l'autre pair est mise à jour.
   (c) Dans $\frac{1}{9}$ des cas, l'algorithme se contente de mettre à jour l'entrée du pair rendez-vous local dans la vue locale des deux pairs.

Comparé au cas 1), le cas 2) est donc moins agressif vis-à-vis des autres pairs rendez-vous. Enfin à chaque itération de l'algorithme, les entrées dont le temps de stockage est supérieur

à PVE_EXPIRATION sont éliminées de la vue locale (ligne 3).

11.3.2 Conditions expérimentales et résultats

L'environnement de test est la grille expérimentale Grid'5000 [51], au maximum 6 sites ont été utilisés (Bordeaux, Lyon, Orsay, Rennes, Sophia-Antipolis et Toulouse). Les tests ont été réalisés en utilisant la version 2.3 de JXTA-C, configurée avec comme protocole de transport TCP et compilée avec gcc 4.0 (niveau d'optimisation -02).

Dans notre test du protocole de gestion de vue, les pairs rendez-vous sont le seul type de pairs utilisé. À chaque fois qu'un pair rendez-vous est ajouté ou retiré de la vue locale d'un pair rendez-vous, le temps écoulé depuis le début du test est sauvegardé dans un fichier, ainsi que le type de l'événement. Nous avons exécuté ce test en utilisant un nombre croissant de pairs rendez-vous, pour plusieurs topologies virtuelles initiales (chaîne, étoile, arbre) et pour différentes valeurs du paramètre PVE_EXPIRATION. L'objectif de ce test est de mesurer le temps nécessaire au protocole de gestion de vue pour que la propriété (11.2) soit satisfaite. Comme nous l'avons vu à la section précédente, une telle propriété assure un routage optimal des requêtes, notamment de découverte, sur le réseau de pairs rendez-vous.

Résultats. La partie gauche de la figure 11.1 présente l'évolution de la taille m de la vue locale d'un pair rendez-vous en fonction du nombre total n de pairs rendez-vous dans le système (n égal à 10, 45, 50, 80, 160 et 580). Pour une même expérience, l'évolution de la valeur de m sur chaque pair rendez-vous appartenant à $\mathbf{S}$ suit la même allure. Nous pouvons constater que si $n \geq 45$, la propriété (11.2) n'est pas satisfaite. Par ailleurs, seulement trois expériences (n égal à 10, 45 et 50) atteignent la taille maximale de la vue locale de $\mathbf{S}$, c'est-à-dire $n - 1$[4] (soit respectivement m égal à 9, 44 et 49). Enfin, l'expérience avec $n = 580$ permet de distinguer nettement trois phases dans la mise en place de la vue locale d'un pair rendez-vous.

1. Augmentation de m. Cette phase se déroule pendant la durée de stockage d'une annonce de pair rendez-vous dans la vue locale d'un pair rendez-vous (paramètre PVE_EXPIRATION, voir section 11.3.1), soit pour l'expérience 20 minutes.
2. Une diminution de m. Elle intervient après, en durée, la valeur du paramètre PVE_EXPIRATION. Si au bout de ce temps, le pair rendez-vous correspondant à cette entrée n'est pas testé à nouveau par l'algorithme de gestion de vue, l'entrée est détruite de la vue locale du pair rendez-vous.
3. Une fluctuation de m autour d'une certaine valeur dépendante de n. Dans notre cas, m oscille autour de 300.

La partie droite de la figure 11.1 montre la distribution des événements d'ajout (représenté par un losange) et de retrait (représenté par une croix) de pairs rendez-vous dans la vue locale d'un pair rendez-vous (pour $n = 580$). Plus précisément, en ordonnée est représenté le numéro du pair rendez-vous[5] de l'expérience considérée. Nous pouvons tout d'abord constater que quasiment tous les pairs rendez-vous sont contactés par chaque pair

[4] Le pair rendez-vous à partir duquel la taille de vue locale est extraite n'est pas comptabilisé dans les mesures.

[5] À chaque apparition dans la vue locale d'un nouvel identifiant, un numéro est attribué en incrémentant une valeur à partir de 1.

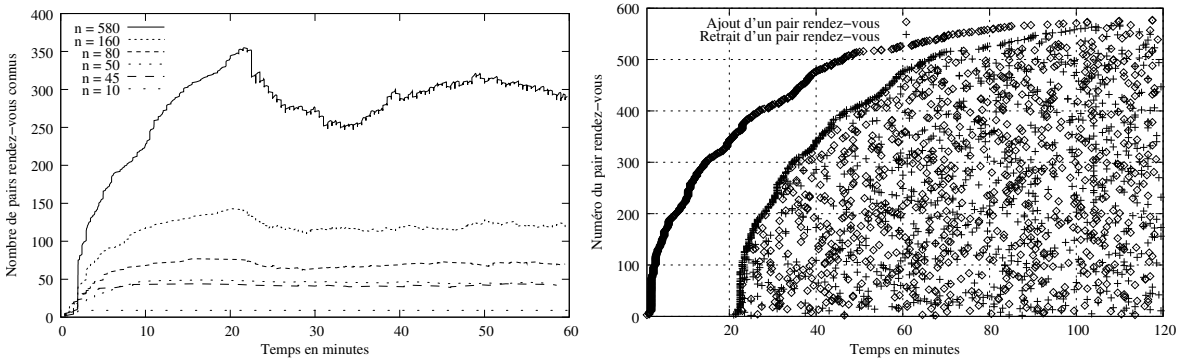

FIG. 11.1 – Évolution de la taille m de la vue locale d'un pair rendez-vous en fonction du nombre total n de pairs rendez-vous dans le système (gauche). Distribution des événements d'ajout et de retrait de pairs rendez-vous de la vue locale d'un pair rendez-vous (droite).

rendez-vous : le numéro maximal de pair rendez-vous atteint est 577 (au bout de 117 minutes), soit 2 manquants. Nous pouvons également constater que cette figure est constituée de deux phases par rapport au temps :

1. Des événements uniquement d'ajout de pairs rendez-vous dans la vue locale du pair rendez-vous. Cette phase dure la valeur du paramètre PVE_EXPIRATION, soit par défaut 20 minutes. Elle commence par une courbe que nous appelons *courbe d'ajouts* (première courbe sur la partie droite de la figure 11.1, constituée de losanges).
2. Des événements d'ajout et de retrait de pairs rendez-vous dans la vue locale du pair rendez-vous. Cette seconde phase dure le temps restant de l'expérience et correspond aux phases 2 et 3 de la partie gauche de la figure 11.1. Elle commence par une courbe que nous appelons *courbe de retraits* (deuxième courbe sur la partie droite de la figure 11.1, constituée de croix).

La partie droite de la figure 11.1 explique la présence de la troisième phase dans la mise en place de la vue locale d'un pair rendez-vous. En effet, la fluctuation de la valeur de m correspond aux retraits et aux ajouts d'entrées dans la vue locale, qui interviennent dans cette seconde phase (partie gauche de la figure 11.1). Elles correspondent à l'incapacité du protocole de gestion de vue de tester toutes les entrées de la vue locale, en un temps inférieur au paramètre PVE_EXPIRATION. Par ailleurs, cette figure explique simplement le fait que la valeur de m culmine dans le temps à la valeur du paramètre PVE_EXPIRATION du protocole de gestion de vue. À partir de cette valeur, les ajouts de pairs rendez-vous (courbe d'ajouts) sont contrebalancés par les retraits de pairs rendez-vous (courbe de retraits). Enfin, notons que la densité en événements de la courbe d'ajouts est plus importante que la densité de la courbe de retraits. Toutefois, la pente instantanée de la première courbe diminue avec l'ajout de pairs rendez-vous, c'est-à-dire que l'algorithme retourne de plus en plus souvent aux pairs locaux des pairs rendez-vous qu'ils connaissent déjà. Par ailleurs, au temps du paramètre PVE_EXPIRATION en minutes, la pente instantanée de la courbe de retraits est nettement supérieure à celle de la courbe d'ajouts.

Discussion. Ce test permet de caractériser le comportement de l'algorithme de gestion de vue de JXTA-C. Il permet notamment une réponse fiable à la question suivante : quel

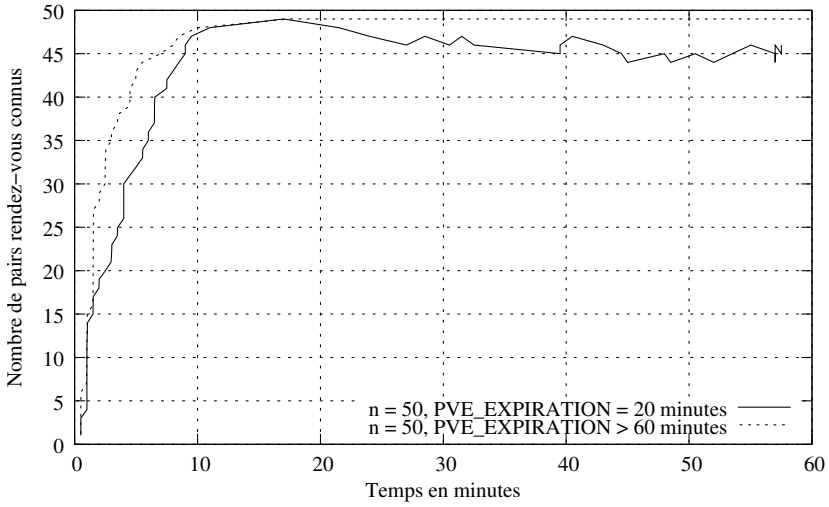

FIG. 11.2 – Évolution de la taille m de la vue locale d'un pair rendez-vous, pour un système P2P composé de 50 pairs rendez-vous, en fonction de la valeur du paramètre PVE_EXPIRATION.

nombre de pairs rendez-vous doit être déployé au sein d'un groupe JXTA ? Cette question est récurrente sur les listes de diffusions de JXTA et jusqu'à présent sans réponse accompagnée de preuves. Ainsi, l'un des concepteurs de cet algorithme[6] nous a affirmé que des valeurs de n variant entre 100 et 150 sont supportées. Notre évaluation montre que dès que n est supérieur à 45, la propriété (11.2) que cherche à remplir cet algorithme n'est pas assurée. En conséquence, avec les valeurs par défaut des paramètres de l'algorithme de gestion de vue, un groupe JXTA composé de 45 pairs rendez-vous n'est pas en mesure de s'organiser au mieux. Ainsi, le routage optimal des requêtes n'est pas garanti, conduisant à une augmentation du nombre de sauts nécessaires pour une requête de découverte. Les paramètres de l'algorithme de gestion de vue doivent être adaptés à la taille totale du nombre de pairs rendez-vous souhaité.

Une solution consiste à modifier la valeur du paramètre PVE_EXPIRATION, comme le montre l'expérience suivante. La figure 11.2 présente l'évolution de la valeur de m sur un pair rendez-vous pour un système à $n = 50$, en fonction de la valeur du paramètre PVE_EXPIRATION. Nous pouvons constater qu'en changeant la valeur de ce paramètre pour un temps supérieur à la durée de l'expérience (60 minutes dans notre cas), la valeur de m atteint sa valeur maximale $n - 1$, soit dans notre cas 49. Par rapport à la propriété 11.2, $t_1 = 17$ minutes. Une autre solution pour supporter un nombre de pairs rendez-vous supérieur à 45 est de diminuer l'intervalle de temps entre les itérations de l'algorithme (paramètre PEERVIEW_INTERVAL, par défaut de 30 secondes). Enfin, une dernière solution est de jouer sur ces deux paramètres à la fois.

Dans les deux cas, un compromis est à trouver entre fraîcheur et donc fiabilité des informations et consommation de la bande passante réseau. La fraîcheur des informations est diminuée lorsque la valeur du paramètre PVE_EXPIRATION est augmentée, alors que le trafic réseau est augmenté lorsque la valeur du paramètre PEERVIEW_INTERVAL est dimi-

[6]Mike Duigou (Sun Mmicrosystems) avec lesquels ce travail a été réalisé en collaboration.

nuée.

11.4 Discussion et conclusion

Dans ce chapitre, nous avons tout d'abord présenté le cadre de ce travail : l'utilisation d'une approche P2P sur les grilles de calcul. Dans ce contexte, le problème du déploiement à grande échelle d'intergiciels pour les grilles de calcul, basés sur des bibliothèques P2P se pose. En effet, il s'agit d'un cas d'utilisation inhabituelle des systèmes P2P qui nécessite un déploiement automatique et non manuel (cas habituel sur Internet). Ce problème constitue en outre un frein à l'utilisation des grilles de calcul comme environnement de test pour la mise au point et l'évaluation des mécanismes P2P à grande échelle. Afin de remédier à ce problème, nous avons ensuite introduit notre contribution : le langage JDL de description de réseaux virtuels JXTA, et son implémentation au sein de l'outil de déploiement générique ADAGE. Ensuite, nous avons présenté la validation de notre proposition à travers l'évaluation à grande échelle d'un protocole de la spécification JXTA (jusqu'à 580 machines réparties sur 8 sites de la grille expérimentale Grid'5000). Cette évaluation a été rendue possible grâce à la mise en œuvre de JDL dans ADAGE, et démontre ainsi son utilité.

Notre évaluation a consisté en des déploiements de réseaux virtuels JXTA composés uniquement de n pairs rendez-vous, n variant de 10 à 580. Ils ont permis de caractériser le comportement de l'algorithme de gestion de vue, notamment de discuter la valeur à choisir pour un paramètre de cet algorithme selon la valeur n. Ainsi, ces tests ont permis d'apporter une réponse fiable à la question suivante : quel nombre de pairs rendez-vous est supporté au sein d'un groupe JXTA ? Cette question est récurrente sur les listes de diffusions de JXTA et jusqu'à présent sans réponse. Des tests à cette échelle d'un système JXTA n'avaient pour l'instant jamais été publiés. Par ailleurs, des mesures préliminaires de performances du protocole de découverte ont été obtenues. Enfin, notons également que le greffon JXTA pour ADAGE a été validé par son utilisation pour le déploiement à grande échelle de réseaux de pairs JUXMEM (voir chapitre 8).

Notre proposition de langage JDL pour la description de réseaux virtuels JXTA s'appuie sur la notion de profil. Elle reste ainsi liée aux types de pairs que définissent les implémentations JXTA-C et JXTA-J2SE de la spécification JXTA. Toutefois, d'autres choix sont possibles, comme par exemple une description en termes de groupe et de service s'exécutant dans chaque groupe. Ils ont par exemple été évoqués sur les liste de diffusion de JXTA. Une telle description permettrait de mieux modéliser la gestion du cycle de vie des services et de leurs dépendances. Elle peut se comparer aux modèles d'empaquetage et de déploiement mis en place par les modèles composant. Toutefois, elle nécessite une révision complète des implémentations actuelles, tâche non triviale.

Cinquième partie

Conclusion et perspectives

Chapitre 12

Conclusion et perspectives

Conclusion

Contexte d'étude. Les grilles de calcul promettent de répondre aux besoins de calcul et de stockage des applications, notamment scientifiques. En effet, ces infrastructures visent à réunir les capacités des ressources informatiques appartenant à des institutions différentes. Ces ressources sont généralement des systèmes parallèles, tels que des supercalculateurs et/ou des grappes de machines. Elles sont interconnectées par des réseaux longue distance à haut débit afin de former un système réparti à grande échelle, hétérogène. Compte tenu de l'échelle visée par une grille de calcul, les machines qui la composent sont sujettes à des conditions de volatilité. L'objectif des concepteurs de ces infrastructures est d'offrir une utilisation *transparente* des grilles de calcul malgré ces contraintes. Ainsi, à terme, une application devra pouvoir être exécutée sur plusieurs sites d'une grille, en la soumettant à travers une banale « prise grille ».

Cette vision idéale n'est toutefois pas encore réalisée. L'émergence des grilles de calcul pose des défis difficiles. Comme nous l'avons vu, de nombreuses contraintes existent et forment un obstacle à la gestion efficace et transparente des réseaux, des processeurs, des espaces de stockage, etc. Pour résoudre ces multiples problèmes, différents modèles de programmation ont été proposés, tels que le modèle Grid-RPC et les modèles à base de composants. Toutefois, aucun de ces modèles de programmation n'offre de solution simple et efficace pour le partage cohérent de données.

Les solutions habituellement utilisées reposent sur le protocole GridFTP de Globus et une gestion manuelle de la localisation des données, de leur transfert, de la tolérance aux fautes, de la création de copies pour optimiser les temps d'accès, etc. Pour ce faire, elles s'appuient sur un ensemble de couches logicielles, utilisant généralement des catalogues de données. Chacun de ces systèmes doit être adapté aux différentes applications. En outre, la cohérence des données partagées entres les différents processus applicatifs est laissée à la charge de l'utilisateur. Dans ce cas, l'utilisateur doit donc *explicitement* gérer à la fois la localisation et la cohérence des données. Une autre approche existe : elle consiste à fournir une interface de type système de fichiers aux applications. On parle alors de systèmes de fichiers

pour grilles de calcul. Cette approche, plus intégrée et offrant une interface standard, propose généralement un modèle de cohérence à la NFS. Un tel modèle n'est cependant pas toujours satisfaisant pour certaines applications scientifiques, telles que les applications de couplage de code. Par ailleurs, le modèle à la NFS n'est également pas adapté aux caractéristiques des grilles de calcul, notamment leur aspect hiérarchique en termes d'interconnexion réseau. Dans ce cas, l'utilisateur doit donc *explicitement* gérer la cohérence des données.

Contributions. Nous avons proposé dans cette thèse le concept de *service de partage de données*, qui offre un *modèle d'accès transparent aux données*. L'objectif de notre contribution est de fournir un système de gestion de données adapté aux architectures des grilles de calcul. Notre service de partage de données prend en charge plusieurs éléments :

- une transparence de la localisation des données ;
- une gestion de la cohérence des données ;
- un stockage persistant ;
- une tolérance à la volatilité des équipements de l'infrastructure visée.

Ce service permet donc de décharger l'utilisateur de la gestion de la localisation et de la cohérence des données partagées des applications s'exécutant sur les grilles de calcul. Cette gestion est *automatiquement et de manière transparente* prise en charge par notre service.

Nos sources d'inspiration pour la conception d'un tel service de partage de données proviennent essentiellement de deux types de systèmes : les systèmes pair-à-pair (P2P) et les systèmes à MVP. Ces deux systèmes ont des propriétés complémentaires et des hypothèses de travail opposées. Or, les infrastructures matérielles que nous visons ont des hypothèses intermédiaires par rapport à celles visées par ces deux types de systèmes. Notre objectif par cette approche *hybride* est de conserver à grande échelle les points forts des systèmes à MVP, c'est-à-dire les modèles et les protocoles de cohérence ainsi que la transparence de localisation des données, grâce à une conception fondée sur une approche P2P. Ainsi, notre concept de service de partage de données s'accompagne d'une proposition d'architecture, appelée JUXMEM. Elle est largement basée sur la spécification P2P JXTA et ses implémentations JXTA-J2SE et JXTA-C, notamment pour la découverte des espaces de stockage libres et les communications dynamiques. Notre évaluation des performances en lecture et écriture de JUXMEM montre des débits satisfaisants, atteignant 90 Mo/s sur un réseau Gigabit Ethernet. Enfin, JUXMEM a fait l'objet d'un dépôt à l'Agence pour la Protection des Programmes et est disponible au téléchargement [218].

Le modèle d'accès transparent aux données fourni par notre concept de service de partage de données a été *intégré* dans deux modèles de programmation utilisés pour concevoir des applications s'exécutant sur les grilles de calcul : *le modèle Grid-RPC et les modèles à base de composants*.

Modèle Grid-RPC. L'intégration que nous avons réalisée du modèle transparent d'accès aux données par notre service a été validée par une mise en œuvre applicative complète. En particulier, nous avons validé l'utilisation de JUXMEM au sein de l'intergiciel Grid-RPC DIET et testé cette intégration via l'application Grid-TLSE. Toutefois, nous avons mené des évaluations de performances seulement via une application synthétique par manque de temps. Ces évaluations ont permis de montrer que la performance d'une application nécessitant une gestion de données persistantes est améliorée lorsque JUXMEM est utilisé. Les accès aux données ainsi que la gestion de la cohérence

et de la tolérance aux fautes sont réalisés de manière transparente pour les développeurs d'applications qui suivent le modèle Grid-RPC. La programmation de telles applications s'en trouve ainsi simplifiée.

Modèle à base de composants. Afin d'intégrer dans les modèles à base de composants le modèle d'accès transparent aux données, nous avons introduit la notion de *port de données*. Cette proposition consiste en un ensemble d'interfaces de programmation pour le partage de données. Ces interfaces sont indépendantes de JUXMEM et des modèles à base de composants que nous avons utilisés : CCM et CCA. Cette proposition de ports de données a été implémentée pour le modèle à base de composants CCM au-dessus de JUXMEM. Cependant, elle n'a pas été évaluée par manque de temps. En revanche, nous avons montré les bénéfices en termes de simplicité de programmation apportés par notre proposition. Nous avons par ailleurs montré comment utiliser notre proposition de ports de données pour autoriser le partage de données passées en paramètres d'opérations entre composants.

Par ailleurs, nous avons également réalisé des contributions « secondaires ». Celles-ci sont orthogonales à notre principale contribution. Elles visent à faciliter l'utilisation de JXTA sur les grilles de calcul et d'en améliorer les performances.

Évaluation de performances et optimisations. Nous avons évalué et optimisé les performances des couches de communication de JXTA-C. Nous avons montré que moyennant une configuration adaptée et diverses optimisations, celles-ci sont utilisables pour réaliser des transferts de données efficaces sur les grilles de calcul, notamment grâce à l'utilisation de la plate-forme haute performance de communication pour grilles PadicoTM. Cette contribution a permis d'améliorer les performances de JUXMEM pour la lecture et l'écriture des données.

Déploiement d'applications JXTA sur les grilles de calcul. Nous avons également contribué à faciliter le déploiement automatique des intergiciels P2P JXTA-J2SE et JXTA-C sur des infrastructures de type grilles de calcul. Nous avons implémenté, sous la forme d'un greffon, le support de JXTA au sein de ADAGE, un logiciel générique de déploiement d'applications sur grilles de calcul. Cette dernière contribution permet de faciliter le déploiement d'intergiciels pour grilles basés sur JXTA, comme JUXMEM par exemple. Elle a été validée à grande échelle par l'évaluation du protocole de gestion de vue de JXTA-C et a également facilité l'évaluation de l'utilisation de JUXMEM par DIET (voir chapitre 8).

Nous tenons à souligner que ces contributions relatives à JXTA ne sont pas spécifiques à JUXMEM. Le choix de JXTA est guidé par son utilisation au sein de JUXMEM, mais nos motivations et notre méthodologie pour ce travail sont indépendantes de la mise en œuvre au sein de JXTA. Ainsi, nos travaux sont susceptibles de faciliter d'autres recherches sur l'utilisation de JXTA et du modèle P2P sur les grilles de calcul. Par ailleurs, ces contributions ont permis de faciliter de déploiement et l'évaluation de JUXMEM, ainsi que d'améliorer ses performances.

Perspectives

Les travaux que nous avons réalisés débouchent sur des prolongements possibles. Dans cette section, nous énumérons ces différentes perspectives. Elles se divisent en deux parties :

d'un côté les perspectives propres à JUXMEM, et de l'autre les perspectives liées à JXTA et son utilisation sur les grilles de calcul.

Perspectives liées à JUXMEM

Interface de type système de fichiers. Les applications scientifiques, de par l'adoption de solutions à base de GridFTP comme pseudo-standard pour la gestion de données sur les grilles, utilisent principalement une interface de type système de fichiers pour accéder et partager des données. Il serait donc intéressant de disposer d'une telle interface de programmation pour JUXMEM. Celle-ci serait directement construite au-dessus de l'interface de programmation de type mémoire de JUXMEM. Cette interface de type système de fichiers, permettrait de monter (en anglais *mount*) JUXMEM comme un banal système de fichiers. Les applications existantes (en anglais *legacy code*) pourraient ainsi bénéficier de manière transparente des fonctionnalités de JUXMEM. Des efforts de développement d'une telle interface ont été initiés avec un module appelé JUXMEMFS. Ce module repose sur le projet FUSE afin que les applications scientifiques puissent utiliser de manière transparente cette interface. Toutefois, ce projet n'est pas encore achevé et n'a pas été évalué.

Cache de données au niveau des clients. Actuellement, dans le modèle de programmation Grid-RPC, une fois qu'un serveur de calcul a utilisé une donnée pour exécuter un appel Grid-RPC d'un client, il la détruit de son espace d'adressage. Au niveau JUXMEM, cela se traduit par exemple par l'utilisation de la primitive juxmem_unmap. Or, dans les applications multiparamétriques par exemple, les données utilisées d'une exécution d'un appel Grid-RPC à une autre diffèrent très peu, voire sont identiques. Ainsi pour la résolution du problème $Ax = b$, il est tout à fait envisageable qu'un même serveur de calcul exécute le problème plusieurs fois sur la même matrice, seuls les paramètres de contrôles des algorithmes utilisés changeant. Ceci conduit donc à implémenter *au niveau des clients* JUXMEM un cache pour stocker ces données. De la sorte, plusieurs appels Grid-RPC successifs sur un même serveur de calcul pourraient réutiliser les données sans nécessiter de communication réseau, à condition d'utiliser les mêmes identifiants de données. Toutefois, cela impliquerait de pouvoir indiquer que certaines données ne sont pas modifiables. Dans le cas contraire, une communication réseau serait nécessaire pour s'assurer de disposer de la dernière version de la donnée.

Gestion des flux de calculs La gestion de données actuellement implémentée dans DIET grâce à l'utilisation de JUXMEM traite de manière indépendante chaque requête de calcul. Cette perspective vise à supporter au niveau de JUXMEM la gestion des données nécessaires pour la mise en œuvre des flux de calculs (en anglais *workflow*). La gestion des flux est actuellement en cours d'implémentation au sein des intergiciels Grid-RPC comme par exemple DIET [17]. L'objectif de cette perspective est d'utiliser les informations des différentes dépendances de données entre les futurs appels Grid-RPC, afin « d'anticiper » et ainsi affiner la gestion des données. Des techniques de préchargement des données sur les serveurs de calculs seraient par exemple intéressantes à étudier. Une interface de programmation d'estimation du temps des transferts de données par JUXMEM serait alors indispensable pour interagir finement avec les ordonnanceurs des intergiciels Grid-RPC et effectuer un placement des calculs en tenant compte de la localisation des données. Par ailleurs et en lien avec

la perspective précédente, la politique du gestion d'un cache au niveau des clients JUXMEM pourrait être finement gérée grâce aux informations de dépendances de données ainsi disponibles.

Données fragmentées et JUXMEM parallèle. Actuellement, JUXMEM copie une donnée vers un client depuis un unique fournisseur, avec éventuellement des flux parallèles grâce à la plate-forme haute performance d'intégration d'exécutifs communicants PadicoTM. Il serait intéressant de fournir une version de JUXMEM dans laquelle les données seraient fragmentées en longueur d'une certaine taille, afin de les distribuer sur plusieurs fournisseurs. L'objectif est de pouvoir gérer des données de grande taille, supérieure à la taille de la mémoire vive d'une machine. Celle-ci est généralement de 2 Go sur les machines utilisées. Cette première amélioration permettrait d'envisager de construire un « JUXMEM parallèle », où les différents fragments d'une donnée seraient écrits/lus en parallèle à partir de plusieurs fournisseurs. L'objectif serait d'agréger la bande passante utilisable pour ces transferts. La conception d'une telle version de JUXMEM, appelée JUXMEM2, nécessiterait certainement des extensions dans l'interface actuelle de programmation de JUXMEM. JUXMEM2 permettrait également d'envisager la conception d'un JUXMEMFS parallèle, en fournissant par exemple une interface respectant la spécification MPI-I/O, de manière analogue à ce que fait le système de fichiers parallèle PVFS.

Utilisation de mécanismes classiques des systèmes à MVP. La version actuelle de JUXMEM bien que largement inspirée par les systèmes à MVP, n'utilise que peu de techniques issues de ces systèmes. Par exemple, il n'y a pas de notion de défaut de page. Il serait donc intéressant d'utiliser les mécanismes de protection de pages pour, par exemple, transférer uniquement les pages nécessaires lors d'accès aux données. Dans ce contexte, la conservation de la sémantique actuelle de la primitive juxmem_attach est problématique. En effet, dans ce cas JUXMEM ne contrôle pas l'adresse de la zone mémoire dont le programmeur souhaite partager les données. Ainsi, celle-ci n'est pas forcément une adresse mémoire d'un multiple d'une page comme le nécessite la primitive mprotect classiquement utilisée pour la mise en œuvre de la notion de défaut de page.

Il serait également intéressant d'offrir une granularité de partage variable pour une donnée. Actuellement, l'utilisation d'un verrou sur une donnée nécessite l'acquisition du verrou pour l'accès à l'ensemble de la donnée. En conséquence, le développeur doit explicitement découper ses données avant de les projeter dans JUXMEM. Cette gestion va à l'encontre de l'objectif de transparence que nous nous sommes fixé. Une solution serait d'offrir une interface de programmation qui permettrait de spécifier la granularité du partage d'une donnée. À terme, selon les schémas d'accès cette granularité pourrait être automatiquement déterminée. À défaut, une certaine granularité peut être fixée.

L'objectif à long terme de cette perspective serait d'aboutir à un système similaire à une MVP, tel que DSM-PM^2, et d'utiliser JXTA pour gérer la dynamicité des communications, la découverte des espaces mémoire, etc. Le nom d'un tel système serait par exemple DSM-JXTA.

Généralisation. Enfin, il serait également intéressant d'étudier l'utilisation de JUXMEM sur d'autres infrastructures que les grilles de calcul. Un exemple de tel cadre est fourni par le pro-

jet Respire[1]. L'objectif de ce projet est de concevoir des environnements P2P avancés pour la gestion de données des applications. Dans le cadre de ce projet, JUXMEM pourrait être utilisé comme plate-forme d'expérimentation pour la mise en œuvre des différentes solutions proposées. Toutefois, les infrastructures visées par le projet Respire, c'est-à-dire Internet, sont bien différentes en termes de caractéristique des infrastructures visées par JUXMEM, à savoir les grilles de calcul. Il serait donc intéressant et nécessaire d'adapter JUXMEM à une utilisation sur Internet. En lien avec la perspective précédente, l'objectif serait de faire de JUXMEM une plate-forme pour le partage de données sur différents types d'infrastructures : que ce soit les grappes de machines, les grilles de calcul ou Internet. À chaque fois, ce métaJUXMEM prendrait une personnalité particulière comme un système à MVP pour les grappes de machines, JUXMEM tel qu'il a été présenté dans ce manuscrit dans le cas des grilles de calcul, etc.

Perspectives liées à l'utilisation de JXTA sur les grilles de calcul

Évaluation et optimisation des intergiciels P2P pour les grilles de calcul. Dans un premier temps, il serait intéressant de poursuivre notre travail d'évaluation des protocoles de JXTA dans le contexte des grilles de calcul. Par exemple, un travail en cours qui fait suite à notre étude du protocole de gestion de vue concerne le protocole de découverte. Cette étude vise à fournir des informations précises sur les performances de ce protocole. De manière plus générale, l'objectif serait de caractériser les performances de l'ensemble des protocoles JXTA.

Par ailleurs, nous avons optimisé les couches de communication de l'intergiciel JXTA-C. Notre objectif était d'utiliser de manière efficace les capacités de réseaux haute performance disponibles sur les grilles de calcul. Dans le prolongement de ce travail, il serait intéressant d'adapter d'autres protocoles de la spécification JXTA aux grilles de calcul. Par exemple, les protocoles d'abonnement aux pairs rendez-vous et de gestion de vue (voir section 6.2.1) pourraient être adaptés pour tenir compte de la différence de latence entre les sites d'une grille. Ainsi, les pairs standards pourraient utiliser cette optimisation pour choisir *automatiquement un pair rendez-vous plus proche physiquement*, et non un au hasard comme c'est le cas actuellement. La bonne localité physique qui en découlerait permettrait par exemple d'améliorer les performances de la propagation des requêtes dans un système P2P basé sur JXTA. À terme, il serait intéressant de fournir des protocoles auto-adaptifs à par exemple la taille du système et aux caractéristiques réseaux des grilles de calcul.

Les optimisations que nous avons réalisées dans ce travail sont bien évidemment spécifiques à JXTA. Toutefois, nous pensons qu'elles sont applicables à d'autres systèmes P2P. En effet, notre démarche mais également nos objectifs ne sont pas liés à JXTA. Il visent à faciliter l'utilisation de techniques P2P sur les grilles de calcul, notamment pour le développement d'intergiciels grilles. Cette utilisation semble indispensable pour, par exemple, le développement d'un système d'exploitation pour les grilles de calcul.

Langage de description de système P2P. Il serait intéressant de réfléchir à intégrer dans notre proposition de langage JDL une description en termes de groupe et de service. L'ob-

[1] Projet de l'appel d'offre Calcul Intensif et Grille de Calcul (CIGC) de l'Agence Nationale pour la Recherche (ANR).

jectif pour un tel langage enrichi serait de servir de langage de description de système P2P, à la manière du langage IDL pour les composants. Il permettrait en outre aux développeurs d'applications, basées sur la spécification JXTA, d'éviter à avoir à écrire une quantité importante de codes de bas niveau, pour démarrer le système, créer des groupes et des services, etc. De plus, ce langage permettrait de mettre en avant la force de JXTA : le fait qu'il s'agisse d'une spécification d'un ensemble de protocoles P2P. En effet, les différentes solutions algorithmiques utilisées, telle l'utilisation de la DHT de FreePastry au lieu de celle de JXTA, pourraient alors être cachées aux développeurs d'applications P2P. Elles seraient automatiquement instanciées par de simples annotations dans ce langage. Dans une autre direction, il serait intéressant de regarder l'utilisation conjointe de langage d'injection de fautes tel que FAIL [96] avec notre langage JDL. L'objectif à terme serait de reproduire des conditions de volatilité réalistes, et d'en mesurer l'impact sur le comportement des protocoles JXTA.

Déploiement dynamique. Enfin, il serait également intéressant de permettre aux systèmes P2P s'exécutant sur les grilles de calcul de s'auto-déployer sur ces infrastructures. Cet auto-déploiement passe par des interactions entre à la fois : l'application utilisatrice de cette fonctionnalité, l'outil de déploiement tel que ADAGE et les systèmes de gestion des ressources des grilles de calcul. Cette fonctionnalité pourrait par exemple être bénéfique à JUXMEM. L'objectif serait de l'adapter à l'exécution aux besoins des applications utilisatrices de ce service. Par exemple, si le système ne dispose pas d'un espace suffisant pour stocker de nouvelles données, JUXMEM pourrait indiquer qu'il souhaite déployer d'autres fournisseurs. À l'inverse et afin d'éviter de consommer inutilement des ressources, des fournisseurs voire des gestionnaires inutiles pourraient être détruits. Poussée à l'extrême, cette idée permettrait à partir d'un unique processus JUXMEM de déployer dynamiquement un système opérationnel complet pour le partage de données. Cette perspective fait partie du sujet de thèse de Loïc Cudennec[2].

[2]Doctorant INRIA/Sun Microsystems au sein du projet PARIS.

Bibliographie

Les premières références bibliographiques correspondent à nos propres publications. Elles sont listées à la page 4.

[14] Marco Aldinucci, Sonia Campa, Massimo Coppola, Marco Danelutto, Domenico Laforenza, Diego Puppin, Luca Scarponi, Marco Vanneschi, and Corrado Zoccolo. Components for high performance grid programming in the Grid.it project. In *Proceedings of the Workshop on Component Models and Systems for Grid Applications*, Saint-Malo, France, June 2004. Held in conjunction with the 2004 ACM International Conference on Supercomputing (ICS '04).

[15] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, Steven Tuecke, and Ian Foster. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. In *Proceedings of the 18th IEEE Symposium on Mass Storage Systems (MSS 2001), Large Scale Storage in the Web*, page 13, Washington, DC, USA, 2001. IEEE Computer Society.

[16] Jim Almond and Dave Snelling. UNICORE: uniform access to supercomputing as an element of electronic commerce. *Future Generation Computer Systems*, 15(5-6):549–558, October 1999.

[17] Abelkader Amar, Raphaël Bolze, Aurélien Bouteiller, Pushpinder Kaur Chouhan, Andréea Chis, Yves Caniou, Eddy Caron, Holly Dail, Benjamin Depardon, Frédéric Desprez, Jean-Sébastien Gay, Gaël Le Mahec, and Alan Su. DIET: New Developments and Recent Results. In *CoreGRID Workshop on Grid Middleware (in conjunction with Euro-Par 2006)*, Dresden, Germany, August 28-29 2006.

[18] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[19] Michele Amoretti, Gianni Conte, Monica Reggiani, and Francesco Zanichelli. Service Discovery in a Grid-based Peer-to-Peer Architecture. In *International Workshop on e-Business and Model Based IT Systems Design*, Saint Petersburg, Russia, April 2004.

[20] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[21] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID '04)*, pages 4–10, Pittsburgh, PA, USA, November 2004. IEEE Computer Society.

[22] Gabriel Antoniu. *DSM-PM2 : une plate-forme portable pour l'implémentation de protocoles de cohérence multithreads pour systèmes à mémoire virtuellement partagée*. Thèse de doctorat, École Normale Supérieure de Lyon, LIP, Lyon, France, novembre 2001.

[23] Gabriel Antoniu, Luc Bougé, and Sébastien Lacour. Making a DSM Consistency Protocol Hierarchy-Aware: an Efficient Synchronization Scheme. In *Proceedings of the Workshop on Distributed Shared Memory on Clusters (DSM 2003)*, pages 516–523, Tokyo, May 2003. Held in conjunction with the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '03).

[24] Luciana Bezerra Arantes, Pierre Sens, and Bertil Folliot. An effective logical cache for a clustered LRC-based DSM system. *Cluster Computing Journal*, 5(1):19–31, January 2002.

[25] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott R. Kohn, Lois C. McInnes, Steve R. Parker, and Brent A. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation (HPDC 8)*, page 13, Redondo Beach, CA, USA, August 1999. IEEE Computer Society.

[26] Dorian C. Arnold, Sudesh Agrawal, Susan Blackford, Jack Dongarra, Micelle Miller, Kiran Sagi, Zhiao Shi, and Sthish Vadhiyar. Users' Guide to NetSolve V1.4. Technical Report CS-01-467, University of Tennessee, Computer Science Dept. Knoxville, TN, USA, July 2001.

[27] Dorian C. Arnold, Dieter Bachmann, and Jack Dongarra. Request Sequencing: Optimizing Communication for the Grid. In *Proceedings of the 6th International Euro-Par Conference (Euro-Par 2000)*, volume 1900 of *Lecture Notes in Computer Science*, pages 1213–1222, Munich, Germany, August 2000. Springer.

[28] Dorian C. Arnold, Sathish S. Vah, and Jack Dongarra. On the Convergence of Computational and Data Grids. *Parallel Processing Letters*, 11(2-3):187–202, June 2001.

[29] Olivier Aumage. *Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes*. Thèse de doctorat, École Normale Supérieure de Lyon, LIP, Lyon, France, septembre 2002.

[30] Didier Badouel, Kadi Bouatouch, and Thierry Priol. Ray Tracing on Distributed Memory Parallel Computers: Strategies for Distributing Computation and Data. *IEEE Computer Graphics and Application*, 14(4):69–77, July 1994.

[31] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer, January 2006.

[32] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC Storage Resource Broker. In *Proceedings of 16th Annual International Conference on Computer Science and Software Engineering (CASCON'98)*, Toronto, Canada, October 1998. ACM Press.

[33] Alessandro Bassi, Micah Beck, Graham Fagg, Terry Moore, James Plank, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: A Study in Resource Sharing. *Future Generation Computer Systems*, 19(4):551–562, 2003.

[34] Alessandro Bassi, Micah Beck, Terry Moore, and James S. Plank. The Logistical Backbone: Scalable Infrastructure for Global Data Grids. In *Proceedings of the 7th Asian Computing Science Conference (ASIAN 2002)*, volume 2550 of *Lecture Notes in Computer Science*, pages 1–12, Hanoi, Vietnam, 2002. Springer.

[35] Alessandro Bassi, Micah Beck, James S. Plank, and Rich Wolski. Internet Backplane Protocol : API 1.0. Technical Report UT-CS-01-455, University of Tennessee, Computer Science Department, Knoxville, TN, USA, March 2001.

[36] Micah Beck, Ying Ding, Erika Fuentes, and Sharmila Kancherla. An Exposed Approach to Reliable Multicast in Heterogeneous Logistical Networks. In *Workshop on Grid and Advanced Networks (GAN '03)*, page 526, Tokyo, Japan, 2003. Held in conjunction with the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '05), IEEE Computer Society.

[37] Micah Beck, Ying Ding, Terry Moore, and James S. Plank. Transnet Architecture and Logistical Networking for Distributed Storage. In *Workshop on Scalable File Systems and Storage Technologies (SFSST)*, San Francisco, CA, USA, September 2004. Held in conjunction with the 17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004).

[38] Micah Beck, Terry Moore, and James S. Plank. An end-to-end approach to globally scalable network storage. In *Proceedings of the 2002 Conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '02)*, pages 339–346, New York, NY, USA, 2002. ACM Press.

[39] Micah Beck, Terry Moore, James S. Plank, and Martin Swany. Logistical Networking: Sharing More Than the Wires. In *Proceedings of 2nd Annual Workshop on Active Middleware Services*, volume 583 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, August 2000.

[40] John Bent, Venkateshwaran Venkataramani, Nick LeRoy, Alain Roy, Joseph Stanley, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. Flexibility, Manageability, and Performance in a Grid Storage Appliance. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC 11)*, pages 3–12, Edinburgh, Scotland, UK, July 2002. IEEE Computer Society.

[41] Brian N. Bershad, Mattew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE International Computer Conference (COMPCON)*, pages 528–537, Los Alamitos, CA, USA, February 1993.

[42] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the 6th workshop on I/O in parallel and distributed systems (IOPADS '99)*, pages 78–88, Atlanta, GA, USA, 1999. ACM Press.

[43] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Néri, and Anton Selikhov. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC '02)*, pages 1–18, Baltimore, Maryland, USA, November 2002. IEEE Computer Society.

[44] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the 7th International Workshop on Component-Oriented Programming (WCOP '02)*, Malaga, Spain, June 2002.

[45] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal Component Model version 2.0-3. Technical report, ObjectWeb Consortium, February 2004.

[46] Jean-Michel Busca, Fabio Picconi, and Pierre Sens. Pastis: A Highly-Scalable Multi-user Peer-to-Peer File System. In *Proceedings of the 11th International Euro-Par Conference (Euro-Par 2005)*, volume 3648 of *Lecture Notes in Computer Science*, pages 1173–1182, Lisbon, Portugal, August 2005. Springer.

[47] Kris Buytaert and Matthias Rechenburg. OpenMosix, Past Present and Future. In *4th International System Administration and Network Engineering Conference (SANE '04)*, Amsterdam, The Netherlands, September 2004. The Netherlands UNIX User Group (NLUUG).

[48] Rajkumar Buyya and Srikumar Venugopal. The Gridbus Toolkit for Service Oriented Grid and Utility Computing: An Overview and Status Report. Technical Report GRIDS-TR-2004-2, Grid Computing and Distributed Systems Laboratory, University of Melbourne, April 2004.

[49] Olof Bärring, Ben Couturier, Jean-Damien Durand, Emil Knezo, Sébastien Ponce, and Vitaly Motyakov. Storage Resource Sharing with CASTOR. In *Proceedings of the 21nd IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '04)*, pages 345–360, Adelphi, MA, USA, April 2004.

[50] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '05)*, pages 776–783, Cardiff, UK, May 2005. IEEE Computer Society.

[51] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid '05)*, pages 99–106, Seattle, Washington, USA, November 2005.

[52] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Hérault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky. Computing on Large Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. *Future Generation Computer Science (FGCS)*, 21(3):417–437, March 2005.

[53] Eddy Caron, Pushpinder Kaur Chouhan, and Holly Dail. GoDIET: A Deployment Tool for Distributed Middleware on Grid'5000. In *Workshop on Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Applications and Tools (EXPGRID)*, pages 1–8, Paris, France, June 2006. In conjunction with 15th IEEE International Symposium on High Performance Distributed Computing (HPDC 15), IEEE Compuer Society.

[54] Eddy Caron and Frédéric Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[55] John B. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, 29:219–227, September 1995. Special issue on Distributed Shared Memory.

[56] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[57] DeQing Chen, Chunqiang Tang, Brandon Sanders, Sandhya Dwarkadas, and Michael L. Scott. Exploiting high-level coherence information to optimize distributed shared state. In *Proceedings of the 9th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)*, pages 131–142, San Diego, CA, USA, June 2003. ACM Press.

[58] Ann Chervenak, Ewa Deelman, Ian Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Kunszt, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggle: a framework for constructing scalable replica location services. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*, pages 1–17, Baltimore, Maryland, 2002. IEEE Computer Society.

[59] Steve T. Chiang, Joseph S. Lee, and Hideaki Yasuda. Data Link Switching Client Access Protocol. IETF Request For Comment 2114, Network Working Group, 1997.

[60] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, July 2003.

[61] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[62] Kark Czajkowski, Donald F. Ferguson, Ian Foster, Jeffery Frey, Steve Graham, Igor Sedukhin, David Snelling, Steven Tuecke, and William Vambenepe. The WS-Resource Framework. Version 1.0, March 2004.

[63] Kark Czajkowski, Donald F. Ferguson, Ian Foster, Jeffrey Frey, Steve Graham, Tom Maguire, David Snelling, and Steven Tuecke. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring and Extension. Version 1.1, March 2004.

[64] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[65] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, number 2735 in Lecture Notes in Computer Science, pages 33–44, Berkeley, CA, USA, February 2003. Springer.

[66] Timothy A. Davis and Iain S. Duff. An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization. *SIAM Journal Matrix Analysis Applications*, 18(1):140–158, 1997.

[67] Michel Daydé, Luc Giraud, Montse Hernandez, Jean-Yves L'Excellent, Chiara Puglisi, and Marc Pantel. An Overview of the GRID-TLSE Project. In *Poster Session of 6th International Conference High Performance Computing for Computational Science (VECPAR '04)*, volume 3402 of *Lecture Notes in Computer Science*, Valencia, Spain, June 2004. Springer.

[68] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Journal of Scientific Programming*, 13(3), 2005.

[69] Thomas A. DeFanti, Ian Foster, Michael E. Papka, Rick Stevens, and Tim Kuhfuss. Overview of the I-WAY: Wide Area Visual Supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2):123–131, 1996.

[70] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM Journal on Matrix Analysis Applications*, 20(3):720–755, 1999.

[71] Alexandre Denis. *Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, décembre 2003.

[72] Frédéric Desprez and Emmanuel Jeannot. Improving the GridRPC Model with Data Persistence and Redistribution. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing (ISPDC '04)*, pages 193–200, Cork, Ireland, July 2004. IEEE Computer Society.

[73] Andrea Domenici, Flavia Donno, Gianni Pucciani, and Heinz Stockinger. Relaxed Data Consistency with CONStanza. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '06)*, pages 425–429, Singapore, May 2006. IEEE Computer Society.

[74] Flavia Donno, Antonia Ghiselli, Luca Magnoni, and Riccardo Zappi. StoRM: grid middleware for disk resource management. In *Proceedings of the 2004 Conference on Computing in High Energy and Nuclear Physics (CHEP '04)*, Interlaken, Switzerland, September 2004. Science Press.

[75] Niels Drost, Rob van Nieuwpoort, and Henri E. Bal. Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing. In *Proceedings of the 6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '06)*, May 2006.

[76] Dirk Düllmann, Wolfgang Hoschek, Francisco Javier Jaén-Martínez, Ben Segal, Heinz Stockinger, Kurt Stockinger, and Asad Samar. Models for Replica Synchronisation and Consistency in a Data Grid. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 10)*, pages 67–75, San Francisco, CA, USA, August 2001. IEEE Computer Society.

[77] F. Dunno, L. Gaido, A. Gishelli, F. Prelz, and M. Sgaravatoo. DataGrid Prototype 1. EU-DataGrid Collaboration. In *Proceedings of the TERENA Networking Conference*, Limerick, Ireland, June 2002.

[78] Alasdair Earl and Phil Clark. Mass Storage Management and the Grid. In *Proceedings of the 2004 Conference on Computing in High Energy and Nuclear Physics (CHEP '04)*, Interlaken, Switzerland, September 2004. Science Press.

[79] Aurélien 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 1, LaBRI, Bordeaux, France, décembre 2005.

[80] Bruno Del Fabbro. *Contribution à la gestion des données dans les grilles de calcul à la demande : de la conception à la normalisation*. Thèse de doctorat, Université de Franche Comté, LIFC, Besançon, France, novembre 2005.

[81] Message Passing Interface Forum. Message Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, May 1994.

[82] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, TN, USA, May 1997.

[83] Ian Foster. What is the Grid? A three-point checklist. *GRIDtoday*, 1(6), July 2002. http://www.gridtoday.com/02/0722/100136.html.

[84] Ian Foster and Adriana Iamnitchi. On Death, Taxes, and the Convergence on Peer-to-Peer and Grid Computing. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, number 2735 in Lecture Notes in Computer Science, pages 118–128, Berkeley, CA, USA, February 2003. Springer.

[85] Ian Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[86] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High PerPformance Computing*, 11(2):115–128, Summer 1997.

[87] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.

[88] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[89] Pierre Fraigniaud and Philippe Gauron. Brief announcement: an overview of the content-addressable network D2B. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing (PODC 2003)*, page 151, Boston, Massachusetts, USA, July 2003. ACM.

[90] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing (HPDC 10)*, pages 55–66, San Francisco, CA, USA, August 2001. IEEE Computer Society.

[91] Nathalie Furmento, Anthony Mayer, Stephen McGough, Steven Newhouse, Tony Field, and John Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.

[92] Andrew S. Grimshaw, William A. Wulf, and the Legion team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 1(40):39–45, January 1997.

[93] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[94] Bryan K. Hess, Michael Haddox-Schatz, and M. Andrew Kowalski. The Design and Evolution of Jefferson Lab's Jasmine Mass Storage System. In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '05) Information Retrieval from Very Large Storage Systems*, pages 94–105, Monterey, CA, USA, April 2005. IEEE Computer Society.

[95] Dean Hildebrand and Peter Honeyman. Exporting Storage Systems in Scalable Manner with pNFS. In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '05) Information Retrieval from Very Large Storage Systems*, pages 18–27, Monterey, CA, USA, April 2005. IEEE Computer Society.

[96] William Hoarau and Sébastien Tixeuil. A language-driven tool for fault injection in distributed system. Research report 1399, Laboratoire de Recherche en Informatique (LRI), University Paris-Sud, France, February 2005.

[97] Jeff Hodges and Robert Morgan. Lightweight Directory Access Protocol (v3): Technical Specification. IETF Request For Comment 3377, Network Working Group, 2002.

[98] K. Holtman, J. Amundson, and P. Avery et al. CMS Requirements for the Grid. In *Proceedings of the 2001 Conference on Computing in High Energy Physics (CHEP '01)*, Beijing, China, September 2001. Science Press.

[99] Peter Honeyman, Wandros A. Adamson, and Shawn McKee. GridNFS: global storage for global collaborations. In *Proceedings of the IEEE International Symposium Global Data Interoperability - Challenges and Technologies*, pages 111–115, Sardinia, Italy, June 2005. IEEE Computer Society.

[100] Wolfgang Hoschek, Javier Jean-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (Grid '00)*, volume 1971 of *Lecture Notes in Computer Science*, pages 77–90, Bangalore, India, December 2000. Springer. Distinguished Paper Award.

[101] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Theory Computing Systems*, 31(4):451–473, 1998. Special Issue: ACM Symposium on Parallel Algorithms and Architectures (SPAA 1996).

[102] Yvon Jégou. Implementation of Page Management in Mome, a User-Level DSM. In *Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid '03)*, pages 479–486, Tokyo, Japan, May 2003. IEEE Computer Society.

[103] Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-optimal Hash Table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, volume 2735 of *Lecture Notes in Computer Science*, pages 98–107, Berkeley, CA, USA, February 2003. Springer.

[104] Nick Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[105] Anne-Marie Kermarrec. *Une approche globale fondée sur la réplication pour la disponibilité et l'efficacité des systèmes extensibles à mémoire partagée*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, octobre 1996.

[106] George Kola, Tevfik Kosar, and Miron Livny. Profiling Grid Data Transfer Protocols and Servers. In *Proceedings of the 10th International Euro-Par Conference (Euro-Par 2004)*, number 3149 in Lecture Notes in Computer Science, pages 452–4593, Pisa, Italy, August 2004. Springer.

[107] George Kola, Tevfik Kosar, and Miron Livny. Run-time adaptation of grid data placement jobs. *Scalable Computing: Practice and Experience*, 6(3):33–43, September 2005.

[108] George Kola and Miron Livny. Diskrouter: A Flexible Infrastructure for High Performance Large Scale Data Transfers. Technical Report CS-TR-2003-1484, University of Wisconsin-Madison Computer Science Department, Madison, WI, USA, 2003.

[109] Tevfik Kosar and Miron Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS '04)*, pages 342–349, Tokyo, Japan, March 2004. IEEE Computer Society.

[110] Klaus Krauter and Muthucumaru Maheswaran. Architecture for a grid operating system. In *Prooceedings of the 1st IEEE/ACM International Workshop on Grid Computing*, volume 1971 of *Lecture Notes in Computer Science*, pages 65–76, Bangalore, India, December 2000. Springer.

[111] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of the 9th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 2000)*, number 2218 in Lecture Notes in Computer Science, pages 190–201, Cambridge, MA, USA, November 2000. Springer.

[112] Gary Kumfert, Tamara Dahlgren, and Thomas Epperly. Babel: A Language Interoperability Tool for Scientific Computing. In *Proceedings of the 8th IBM System Scientific Computing User Group (SCICOMP 8)*, Minneapolis, USA, August 2003.

[113] Peter Z. Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. File-based replica management. *Future Generation Computing Systems*, 21(1):115–123, 2005.

[114] Sébastien Lacour. *Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, décembre 2005.

[115] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[116] Philippe Lebrun. The Large Hadron Collider, a Megascience Project. In *38th INFN Eloisatron Project Workshop on Superconducting Materials for High Energy Colliders*, Erice, Italy, October 1999.

[117] Kai Li. *Shared virtual memory on loosely coupled multiprocessors*. Phd thesis, Yale University, 1986.

[118] Kai Li. IVY: a shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 94–101, University Park, PA, USA, August 1988. Pennsylvania State University Press.

[119] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[120] Tyng-Yeu Liang, Chun-Yi Wu, Jyh-Biau Chang, Ce-Kuen Shieh, and Pei-Hsin Fan. Enabling Software DSM System for Grid Computing. In *Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks (ISPAN 2005)*, page 6, Las Vegas, NV, USA, December 2005. IEEE Computer Society.

[121] Jeff Magee, Naranker Dulay, and Jeff Kramer. A Constructive Development Environment for Parallel and Distributed Programs. In *Proceedings of the 2nd International Workshop on Configurable Distributed Systems (IWCDS '02)*, pages 4–14, Pittsburgh, PA, USA, March 1994. IEEE Computer Society Press.

[122] Neil Miller, Rob Latham, Robert B. Ross, and Phil Carns. Improving Cluster Performance with PVFS2. *ClusterWorld Magazine*, 2(4), April 2004.

[123] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Labs, March 2002.

[124] Andrey Mirtchovski, Rob Simmonds, and Ron Minnich. Plan 9 - an integrated approach to grid computing. In *Workshop on High-Performance Grid Computing*, page 273a, Santa Fe, New Mexico, USA, April 2004. Held in conjunction with the 18th International Parallel and Distributed Processing Symposium (IPDPS '04).

[125] Kenichi Miura. Overview of Japanese National Research Grid Initiative (NAREGI) Project. *Fujitsu Scientific & Technical Journal*, 40(2):196–204, December 2004.

[126] Hashim H. Mohamed and Dick H.J. Epema. Experiences with the KOALA Co-Allocating Scheduler in Multiclusters. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '05)*, pages 784–791, Cardiff, UK, May 2005. IEEE Computer Society.

[127] Sébastien Monnet. *Gestion des données dans les grilles de calcul : support pour la tolérance aux fautes et la cohérence des données*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, novembre 2006.

[128] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics Magazine*, 38:114–117, April 1965.

[129] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[130] Hidemoto Nakada, Mitsuhisa Sato, and Satoshi Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems*, 15(5-6):649–658, 1999. Special issue on metacomputing.

[131] Hidemoto Nakada, Yoshio Tanaka, Keith Seymour, Frédéric Desprez, and Craig Lee. The End-User and Middleware APIs for GridRPC. In *Proceedings of the Workshop on Grid Application Programming Interfaces (GAPI '04)*, Brussels, Belgium, September 2004. In conjunction with Global Grid Forum 12 (GGF).

[132] Raymond Namyst and Jean-François Méhaut. *Marcel : une bibliothèque de processus légers*. LIFL, Université Sciences et Technique Lille, France, 1995.

[133] Raymond Namyst and Jean-François Méhaut. PM2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285, Gent, Belgium, September 1995. Elsevier Science Publishers.

[134] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003. Special issue of the 3rd Workshop on Run-time Verification (RV '2003).

[135] Ralph Niederberger. DEISA: Motivations, Strategies, Technologies. In *Proceedings of the International Supercomputer Conference (ISC '04)*, Heidelberg, Germany, June 2004.

[136] Open Management Group (OMG). CORBA Components, Version 3. Document formal/02-06-65, OMG, June 2002.

[137] Open Management Group (OMG). The Common Object Request Broker: Architecture and Specification V3.0. Document formal/02-06-33, OMG, June 2002.

[138] Andy Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Gnutella, pages 94–122. O'Reilly, May 2001.

[139] Pradeeps Padala and Joseph N. Wilson. GridOS: Operating system services for grid architectures. In *Proceedings of International Conference on High Performance Computing (HiPC '03)*, volume 2913 of *Lecture Notes in Computer Science*, pages 353–362, Hyderabad, India, December 2003. Springer.

[140] Rob Pennington. Terascale Clusters and the TeraGrid. In *Proceedings of 6th International Conference/Exhibition on High Performance Computing in Asia Pacific Region*, pages 407–413, Bangalore, India, December 2002. Invited talk.

[141] Timur Perelmutov, John Bakken, and Don Petravick. Storage Resource Manager. In *Proceedings of the 2004 Conference on Computing in High Energy and Nuclear Physics (CHEP '04)*, Interlaken, Switzerland, September 2004. Science Press.

[142] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrean, Ken Thompson, Phil Trickey, and Howard Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[143] James Plank, Micah Beck, Wael Elwasif, Terence Moore, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: Storage in the network. In *Network Storage Symposium (NetStore '99)*, Seattle, WA, October 1999.

[144] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 311–320, Newport, RI, USA, June 1997.

[145] Christian Pérez, Thierry Priol, and André Ribes. A Parallel CORBA Component Model for Numerical Code Coupling. *The International Journal of High Performance Computing Applications (IJHPCA)*, 17(4):417–429, 2003. Special issue on Best Applications Papers from the 3rd International Workshop on Grid Computing.

[146] Arcot Rajasekar, Michael Wan, Reagan Moore, Wayne Schroeder, George Kremenek, Arun Jagatheesan, Charles Cowart, Bing Zhu, Sheau-Yen Chen, and Roman Olschanowsky. Storage Resource Broker - Managing Distributed Data in a Grid. *Computer Society of India Journal*, 33(4):42–54, October 2003. Special Issue on SAN.

[147] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore Prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, April 2003.

[148] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in DHT. Technical Report UCB/CSD-03-1299, Univeristy of California, Computer Science Division (EECS), Berkeley, CA, USA, December 2003.

[149] Louis Rilling. *Système d'exploitation à image unique pour une grille de composition dynamique : conception et mise en oeuvre de services fiables pour exécuter les applications distribuées partageant des données*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, novembre 2005.

[150] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–250, Heidelberg, Germany, November 2001. Springer.

[151] Antony I. T. Rowstron and Peter Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Alberta, Canada, October 2001.

[152] Asad Samar and Heinz Stockinger. Grid Data Management Pilot (GDMP): A Tool for Wide Area Replication in High-Energy Physics. In *Proceedings of the 19th IASTED International Conference on Applied Informatics (AI '01)*, Innsbruck, Austria, February 2001.

[153] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX Summer Technical Conference*, pages 119–130, Portland, OR, USA, June 1985.

[154] Mitsuhisa Sato, Taisuke Boku, and Daisuke Takahasi. OmniRPC: a Grid RPC System for Parallel Programming in Cluster and Grid Environment. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '03)*, pages 206–213, Tokyo, Japan, May 2003. IEEE Computer Society.

[155] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *Proceedings of the 3rd International Workshop on Grid Computing (GRID '02)*, volume 2536 of *Lecture Notes in Computer Science*, pages 274–278, Baltimore, MD, USA, November 2002. Springer.

[156] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eilser, and David Noveck. Network File System (NFS) version 4 Protocol. IETF Request For Comment 3530, Network Working Group, 2003.

[157] Clay Shirky. What is P2P... and what isn't it? http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html, November 2000.

[158] Arie Shoshani, Alex Sim, and Junmin Gu. Storage Resource Managers: Middleware Components for Grid Storage. In *Proceedings of the 10th NASA Goddard Conference on Mass Storage Systems and Technologies, 19th IEEE Symposium on Mass Storage Systems (MSST '02)*, pages 209–223, College Park, MA, USA, April 2002. IEEE Computer Society.

[159] Alex Sim, Junmin Gu, Arie Shoshani, and Vijaya Natarajan. DataMover: Robust Terabyte-Scale Multi-file Replication over Wide-Area Networks. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM '04)*, pages 403–414, Santorini Island, Greece, June 2004. IEEE Computer Society.

[160] Olivier Soyez. *Stockage dans les systèmes pair à pair*. Thèse de doctorat, Université de Picardie Jules Verne, LaRIA, Amiens, France, novembre 2005.

[161] Evan Speight and John K. Bennett. Brazos: A third generation DSM system. In *Proceedings of 1st USENIX Windows NT Workshop*, pages 95–106, Seattle, Washington, USA, August 1997.

[162] Raj Srinivasan. XDR: External data representation standard. IETF Request For Comment 1832, Network Working Group, 1995.

[163] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM 2001)*, pages 149–160, San Diego, CA, August 2001.

[164] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, pages 149–160, San Diego, CA, USA, 2001.

[165] Vaidy Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[166] Martin Swany. Improving Throughput for Grid Applications with Network Logistics. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, page 23, Washington, DC, USA, November 2004. IEEE Computer Society.

[167] Martin Swany and Rich Wolski. Network Scheduling for Computational Grid Environments. *Scalable Computing: Practice and Experience*, 6(3):85–94, September 2005.

[168] Martin Swany and Richard Wolski. Data Logistics in Network Computing: The Logistical Session Layer. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA '01)*, pages 174–185, Cambridge, MA, USA, October 2001.

[169] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[170] Domenico Talia and Paolo Trunfio. Toward a Synergy Between P2P and Grids. *IEEE Internet Computing*, 7(4):94–96, 2003.

[171] Domenico Talia and Paolo Trunfio. A P2P Grid Services-Based Protocol: Design and Evaluation. In *Proceedings of the 10th International Euro-Par Conference (Euro-Par 2004)*, number 3149 in Lecture Notes in Computer Science, pages 1022–1031, Pisa, Italy, August 2004. Springer.

[172] Andrew S. Tanenbaum and Martin van Steen. *Distributed Systems, Principles and Paradigms*. Prentice-Hall, 2002.

[173] Osamu Tatebe, Youhei Morita, Satoshi Matsuoka, Noriyuki Soda, and Satoshi Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '02)*, pages 102–110, Berlin, Germany, May 2002. IEEE Computer Society.

[174] Osamu Tatebe, Noriyuki Soda, Youhei Morita, Satoshi Matsuoka, and Satoshi Sekiguchi. GFarm v2: a grid file system that supports high-performance distributed and parallel data computing. In *Proceedings of the 2004 Conference on Computing in High Energy and Nuclear Physics (CHEP '04)*, Interlaken, Switzerland, September 2004. Science Press.

[175] Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny. The Kangaroo Approach to Data Movement on the Grid. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 10)*, pages 325–333, San Francisco, CA, USA, August 2001. IEEE Computer Society.

[176] Douglas Thain and Miron Livny. Parrot: Transparent User-Level Middleware for Data-Intensive Computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, September 2005.

[177] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[178] Nicolas Théodoloz. DHT-based Routing and Discovery in JXTA. Master's thesis, École Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, February 2004.

[179] Bernard Traversat, Mohamed Abdelaziz, Dave Doolin, Mike Duigou, Jean-Christophe Hugly, and Eric Pouyoul. Project JXTA-C: Enabling a Web of Things. In *36th Annual Hawaii International Conference on System Sciences (HICSS '03)*, page 282b, Big Island, Hawaii, USA, January 2003. IEEE Computer Society.

[180] Bernard Traversat, Mohamed Abdelaziz, and Eric Pouyoul. Project JXTA: A Loosely-Consistent DHT Rendezvous Walker. http://www.jxta.org/docs/jxta-dht.pdf, March 2003.

[181] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA 2.0 Super-Peer Virtual Network. http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf, May 2003.

[182] Steven Tuecke, Kark Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, Carl Kesselman, Tom Maguire, Thomas Sandholm, Peter Vanderbilt, and David Snelling. Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum Draft Recommendation, June 2003.

[183] Amin Vahdat, Thomas Anderson, Michael Dahlin, David Culler, Eshwar Belani, Paul Eastham, and Chad Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedins of the 7th IEEE Symposium on High Performance Distributed Computing*, pages 52–63, Chicago, Illinois, USA, July 1998. IEEE Computer Society.

[184] Geoffroy Vallée, Renaud Lottiaux, Louis Rilling, Jean-Yves Berthou, Ivan Dutka-Malhen, and Christine Morin. A Case for Single System Image Cluster Operating Systems: the Kerrighed Approach. *Parallel Processing Letters*, 13(2):95–112, June 2003.

[185] Srikumar Venugopal, Rajkumar Buyya, and Lyle Winton. A Grid service broker for scheduling e-Science applications on global data Grids: Research Articles. *Concurrencey and Computation: Practice & Experience*, 18(6):685–699, 2006.

[186] Bruce Walker. OpenSSI Linux Cluster Project. Talk at Linux Bangalore. http://linux-bangalore.org/2003/schedules/talkdetails.php?talkcode=B303% , December 2003.

[187] Baohua Wei, Gilles Fedak, and Franck Cappello. Scheduling independent tasks sharing large data distributed with BitTorrent. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid' 05)*, page 8, Seattle, Washington, USA, November 2005. IEEE Computer Society.

[188] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 255–270, Boston, MA, USA, 2002. ACM Press.

[189] Brian S. White, Andrew S. Grimshaw, and Anh Nguyen-Tuong. Grid-Based File Access: The Legion I/O Model. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC 9)*, pages 165–174, Pittsburgh, PA, USA, August 2000. IEEE Computer Society.

[190] Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. LegionFS: a secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)*, pages 59–59, New York, NY, USA, 2001. ACM Press.

[191] Ben Yanbin Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Univeristy of California, Computer Science Division (EECS), Berkeley, CA, USA, April 2001.

[192] *Proceedings of the 1st EGEE Conference*, University College Cork, Ireland, April 2004.

[193] *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, number 2429 in Lecture Notes in Computer Science, Cambridge, MA, USA, March 2002. Springer.

[194] ADSTAR Distributed Storage Manager (ADSM). http://www.almaden.ibm.com/cs/stgsysrv/csadsm.html.

[195] ApGrid project. http://www.apgrid.org/.

[196] Apache Portable Runtime. http://apr.apache.org/.

[197] Avaki EII - Sysbase Inc. http://www.sybase.com/products/developmentintegration/avakieii.

[198] Chirp protocol specification. http://www.cs.wisc.edu/condor/chirp/PROTOCOL.

[199] China National Grid project. http://www.cngrid.org/en_index.htm.

[200] DAIS Working Group. http://forge.gridforum.org/projects/dais-wg.

[201] DataSynapse. http://www.datasynapse.com/.

[202] Data Migration Facility (DMF). http://www.sgi.com/products/storage/tech/dmf.html.

[203] Dolphin Interconnect Solutions Inc. http://www.dolphinics.com/.

[204] Électricité de France. http://www.edf.fr/.

[205] eDonkey 2000. http://www.edonkey2000.com/.

[206] FreePastry. http://freepastry.org/.

[207] Filesystem in Userspace (FUSE). http://fuse.sourceforge.net/.

[208] GDS: Grid Data Service. http://www.irisa.fr/GDS/.

[209] GÉANT2 project. http://www.geant2.net/.

[210] Grid'5000 Project. http://www.grid5000.fr/.

[211] GridLab: A Grid Application Toolkit and Testbed. http://www.gridlab.org/, 2005.

[212] Grid Remote Procedure Call WG (GRIDRPC-WG). https://forge.gridforum.org/projects/gridrpc-wg/.

[213] Grifi: GridFTP File System. http://grifi.sourceforge.net/.

[214] High Performance System Storage (HPSS). http://www.hpss-collaboration.org/.

[215] Le projet HydroGrid. http://www-rocq.inria.fr/~kern/HydroGrid/HydroGrid.html.

[216] InfiniBand Trade Association. http://www.infinibandta.org/.

[217] JXTA Distributed Framework. http://jdf.jxta.org/.

[218] The JuxMem Project: Juxtaposed Memory. http://juxmem.gforge.inria.fr/.

[219] JXTA ext:config project. http://wiki.java.net/bin/view/Jxta/ExtConfig.

[220] Cog Kit JXTA Project. http://www-unix.globus.org/cog/projects/jxta/.

[221] JXTA-J2SE Project. http://platform.jxta.org/.

[222] JXTA Specification project. http://spec.jxta.org/.

[223] Kadeploy. http://www-id.imag.fr/Logiciels/kadeploy/.

[224] KaZaA. http://www.kazaa.com/.

[225] KCachegrind - Profiling Visualization. http://kcachegrind.sourceforge.net/.

[226] Lightweight Data Replicator. http://www.lsc-group.phys.uwm.edu/LDR/.

[227] Load Sharing Facility (LSF). http://www.platform.com/products/LSF/.

[228] The Mico CORBA Component Project. http://www.fpx.de/MicoCCM/.

[229] Moab Grid Suite. http://www.clusterresources.com/pages/products/moab-grid-suite.php.

[230] Myri-10G and Myrinet-2000 Performance Measurements. http://www.myricom.com/scs/performance/.

[231] MySQL. http://www.mysql.com/.

[232] Napster protocol specification. http://opennap.sourceforge.net/napster.txt, March 2001.

[233] Norwegian Grid project. http://www.norgrid.no/.

[234] OARGRID. http://oargrid.gforge.inria.fr/.

[235] The OGSA-DAI Project. http://www.ogsadai.org.uk/.

[236] OpenCCM - The Open CORBA Component Model Platform. http://openccm.objectweb.org/.

[237] Portable Batch System (OpenPBS). http://www.openpbs.org/.

[238] Platform. http://www.platform.com/.

[239] Quadrics. http://www.quadrics.com/.

[240] Renater : Le Réseau National de Télécommunications pour la Technologie, l'Enseignement et la Recherche. http://www.renater.fr/.

[241] RSL Specification 1.0. http://www.globus.org/toolkit/docs/2.4/gram/rsl_spec1.html.

[242] Scilab. http://www.scilab.org/.

[243] Sun Grid Engine (SGE). http://www.sun.com/software/gridware/.

[244] SRBfs. http://www.nbirn.net/Resources/Users/Applications/SRBfs/.

[245] Storage Resource Management Working Group. http://sdm.lbl.gov/srm-wg/.

[246] Swedish Grid project. http://www.swegrid.se/.

[247] Univa Corp. http://www.univa.com/.

[248] XtreemOS. https://www.xtreemos.org/.

Sixième partie

Annexes

A.1 Description complète de l'interface de programmation de JUXMEM

Cette annexe présente en détails l'ensemble des primitives de l'interface de programmation de JUXMEM.

void* juxmem_malloc(size_t size, char **data_id). Cette primitive permet d'allouer un ensemble d'espaces mémoire dans le service de partage de données. En cas de succès, cette primitive retourne un pointeur sur un espace mémoire, d'une taille de size octets, du processus applicatif permettant d'y écrire et/ou lire une donnée partagée. En cas d'échec, cette primitive retourne un pointeur nul. La variante à cette fonction, juxmem_calloc initialise à 0 l'ensemble des espaces mémoire alloués dans le service de partage de données.

void juxmem_free(char *data_id). Cette primitive permet de libérer l'ensemble des espaces mémoire alloués par le service pour le stockage de la donnée identifiée par data_id. La destruction de l'ensemble des copies n'est pas garantie si l'environnement d'exécution est soumis à des conditions de volatilité. Toutefois, l'accès à une copie d'une donnée libérée par cette primitive n'est plus possible. Un réinitialisation du processus qui stocke ce type de copies, dites *orphelines*, est alors nécessaire afin d'éviter la perte d'espace de stockage dans le service.

void* juxmem_mmap(void *ptr, size_t len, char *data_id, off_t off). Cette primitive permet de projeter à l'adresse ptr du processus applicatif les len octets de la donnée identifiée par data_id à partir de l'offset off. Si le pointeur ptr est nul, un nouvel espace mémoire est alloué puis retourné par la primitive. Si len est égal à 0, la totalité de l'espace mémoire est projeté. Si l'identifiant de la donnée n'existe pas dans le service ou si les paramètres len et off sont invalides, la primitive échoue. En cas d'échec, la primitive retourne le pointeur nul.

void juxmem_unmap(void *ptr). Cette primitive détruit la projection effectuée à l'adresse ptr du processus applicatif d'une donnée partagée via le service. En outre, l'espace mémoire local est libéré. Toutefois, notons bien que l'ensemble des espaces mémoire utilisés pour stocker une donnée n'est pas réclamé par le service. Ainsi, la donnée est toujours accessible par d'autres processus applicatifs.

Nous offrons également une primitive supplémentaire appelée juxmem_mmap_offset, utile pour l'accès à des données de grande taille. Son prototype complet est le suivant :

int juxmem_mmap_offset(void **ptr, size_t *len, off_t offset)

Cette primitive permet à un processus qui a projeté dans son espace d'adressage une partie d'une donnée partagée, d'avoir accès au reste de la donnée sans devoir allouer des espaces mémoire locaux supplémentaires. Cette primitive permet à un processus applicatif de changer la valeur du pointeur ptr pour pointer la valeur offset (en octets) de la donnée partagée associée. len indique en entrée la longueur de la donnée que le processus souhaite projeter localement à partir de l'offset offset. En sortie, elle correspond à la taille réellement projetée localement. Si cette taille est inférieure à celle spécifiée en entrée, la taille totale de la donnée partagée a été dépassée. La valeur de la zone ainsi non utilisée n'est pas modifiée. Cette primitive échoue pour des valeurs non valides des paramètres d'entrée.

char∗ juxmem_attach(void ∗ptr, size_t len). Comparée à juxmem_malloc, cette primitive permet à un processus applicatif de partager une donnée déjà allouée localement. En effet, elle permet de projeter les len octets de l'adresse pointée par ptr du processus applicatif dans le service de partage de données. En cas de succès de l'allocation de l'ensemble des espaces mémoire, cette primitive retourne l'identifiant de la donnée partagée. En cas d'échec, elle retourne un identifiant de donnée nul (chaîne de caractères vide).

void juxmem_detach(void ∗ptr). Cette primitive détruit la projection effectuée à l'adresse ptr du processus applicatif d'une donnée partagée via notre service. En revanche, l'espace mémoire local n'est pas libéré et reste donc toujours valide, pour une éventuelle utilisation ultérieure. Ainsi, les éventuelles modifications apportées par d'autres processus applicatifs ne seront pas propagées au processus courant. De manière similaire à juxmem_unmap, l'ensemble des espaces mémoire utilisés pour stocker une donnée n'est pas réclamé par le service.

void∗ juxmem_realloc(void ∗ptr, size_t size). Cette primitive permet de modifier la taille à size octets, de l'ensemble des espaces mémoire associés à l'identifiant de données, dont une copie est localement accessible via le pointeur ptr. Si l'un ou plusieurs des espaces mémoire où est stockée la donnée doivent être déplacés, ils sont libérés puis une nouvelle allocation du nombre nécessaire d'espaces mémoire est effectuée. Si les nouveaux espaces mémoire sont d'une taille supérieure, leurs contenus dans leur nouvelle portion mémoire sont indéfinis. En cas de succès, cette primitive retourne un pointeur sur un espace mémoire, d'une taille size octets, du processus applicatif. En cas d'échec, elle retourne le pointeur nul.

Notons que l'utilisation des primitives juxmem_attach et juxmem_map n'oblige en rien l'utilisation des primitives juxmem_detach et juxmem_unmap respectivement. Tout dépend du processus applicatif (s'il souhaite garder un pointeur local valide sur la donnée ou non).

Les primitives du modèle d'accès mémoire de JUXMEM ont été présentées à la section 5.1.4.2. Nous offrons également la primitive décrite ci-dessous.

int msync(void ∗ptr). Cette primitive permet de propager les modifications apportées localement par le processus applicatif à la donnée partagée qui est stockée localement dans l'espace mémoire pointé par ptr. Son usage est limité aux processus ayant le verrou associé à la donnée en écriture, ou lorsque la donnée partagée n'est projetée que par un unique processus. En cas de non respect de ces conditions, son appel échoue.

A.2 Encore plus sur JXTA !

Cette annexe donne davantage de renseignements sur l'environnement JXTA.

A.2.1 Performances initiales des couches de communications

La figure A.1 présente les performances en termes de débits des couches de communications de JXTA-J2SE 2.3.2 et JXTA-C 2.2. Elles représentent les performances initiales de ces intergiciels telles que nous les avons évaluées.

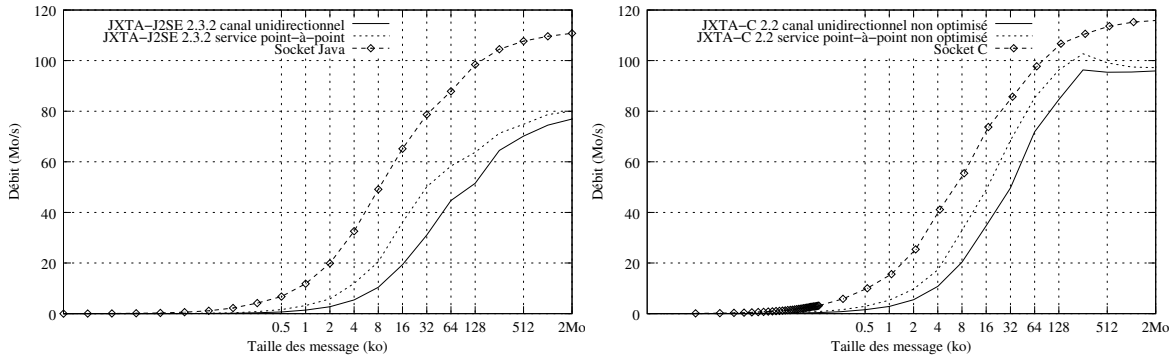

FIG. A.1 – Débit des couches de communications de JXTA-J2SE 2.3.2 non optimisé (gauche) et de JXTA-C 2.2 (droite) sur un réseau SAN Gigabit Ethernet, en prenant pour courbe de référence le débit d'une *socket*.

A.2.2 Algorithmique du service de découverte

Comme nous l'avons vu au chapitre 6, chaque ressource disponible sur un réseau JXTA est décrite sous la forme d'une annonce (voir définition 6.3). Chaque type annonce définit un ensemble d'attributs qui servent lors de la publication d'une annonce de ce type. Chaque attribut est utilisé par le mécanisme d'index distribué des pairs rendez-vous pour constituer un index. Un index correspond au couple suivant :

(SHA-1(expression),identifiant)

L'élément expression est constitué du type de l'annonce, du nom de l'attribut à indexer ainsi que sa valeur. Par exemple, sur la figure A.2 l'annonce à publier est de type Peer, un des attributs à indexer est Name et la valeur de cet attribut est Test. Ainsi, l'élément expression est égal dans ce cas à PeerNameTest. L'élément identifiant correspond à l'identifiant du pair qui publie cette annonce. Les index sont distribués sur l'anneau des pairs rendez-vous en utilisant une table de hachage dite *faiblement couplée*, que nous avons précédemment notée LC-DHT (voir section 6.2.2). Le placement des index s'appuie sur la vue locale du pair rendez-vous auquel le pair qui publie l'annonce est abonné. À cet effet, la fonction suivante est utilisée :

Fonction `ReplicaPeer(`*expression*`)` utilisée pour trouver le pair responsable d'un index d'une annonce.

1 $hachage =$ `SHA-1` (expression);
2 $pos =$ `Valeur entière` (hachage $* \frac{n}{\text{MAX\_HASH}}$);
3 Retourner l'élément de la vue locale situé à la position pos;

La constante MAX_HASH correspond à la valeur maximale que la fonction de hachage peut retourner et n étant toujours le nombre de pairs rendezvous dans la vue locale du pair rendezvous exécutant cette fonction. Enfin, lorsqu'un pair rendez-vous stocke un index d'une annonce, on dit qu'il en devient responsable. Un tel pair s'appelle alors le *pair responsable* (dans le code *replica peer*) pour un attribut de cette annonce.

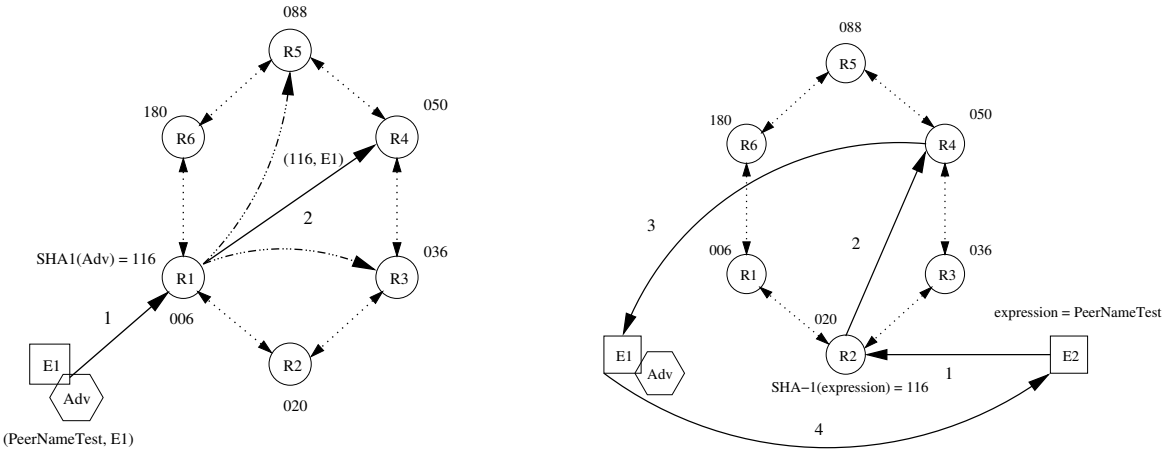

FIG. A.2 – Étapes de la publication (gauche) et de la recherche (droite) d'une annonce sur un réseau de pairs rendez-vous.

Publication d'une annonce. La figure A.2 représente le cheminement d'une requête pour la publication d'une annonce de type pair (Peer), avec comme attribut à indexer Name, la valeur de cet attribut étant Test. Elle se décompose en deux étapes qui sont décrites ci-dessous.

1. Le couple $(\text{PeerNameTest}, E1)$ est donc envoyé sur le pair rendez-vous auquel est abonné le pair standard $E1$. Supposons que la fonction de hachage sur l'expression PeerNameTest retourne la valeur 116. Par ailleurs, supposons que la valeur maximale retournée par cette fonction (MAX_HASH) soit égale à 200. Supposons également que les vues locales des pairs rendez-vous soient convergentes et d'une taille égale à 6. Ainsi, lors de l'appel à la fonction |ReplicaPeer| la valeur de pos est de 3 ($\frac{116 * 6}{200}$). Dans la liste triée de la vue locale du pair rendez-vous $R1$, le pair responsable retourné est donc $R4$ (le premier élément dans la vue locale est considéré comme positionné à 0).
2. Le couple (116, E1) est donc envoyé sur le pair $R4$. Par ailleurs, dans [180] les auteurs de l'algorithme indiquent également que ce couple est envoyé sur le pair de rendez-vous inférieur $R3$ et le pair de rendez-vous supérieur $R5$. Notons toutefois que cette réplication n'est en pratique pas implémentée, ni dans JXTA-C ni dans JXTA-J2SE.

Recherche d'une annonce. La figure A.2 illustre les différentes étapes d'une requête de recherche pour la même annonce que dans l'exemple précédent. Les différentes opérations réalisées pour cette recherche sont décrites ci-dessous.

1. L'expression recherchée PeerNameTest est transmise sur le pair rendez-vous $R2$ auquel est abonné le pair standard $E2$. Cette étape est optionnelle si le pair qui a lancé la recherche est un pair rendez-vous.
2. Le pair $R2$ va alors déterminer que le pair responsable de l'annonce recherchée est $R4$ et lui transmettre la requête. Pour ce faire il va utiliser la même fonction ReplicaPeer que lors de la publication, sur l'expression à rechercher[3].

[3]En faisant l'hypothèse que la requête concerne le même attribut Name.

3. Grâce à un mécanisme de durée de vie associé à la requête de découverte, le pair $R4$ ne va pas calculer le pair responsable pour l'index de l'annonce recherché. En revanche et par l'intermédiaire de son cache local d'index, il va déterminer que le pair qui a publié l'annonce est le pair standard $E1$. Il va donc lui faire suivre la requête.
4. Finalement, ce dernier étant bien le propriétaire de l'annonce recherchée, il va l'envoyer au pair standard $E1$.

Lien avec le protocole de gestion de vue. À la fois lors de la publication et lors de la recherche des annonces, la vue locale des pairs rendez-vous est utilisée pour transmettre les messages vers le pair responsable supposé. Appelons $R1$ le pair responsable d'un attribut d'une annonce, ce pair étant désigné par la fonction ReplicaPeer lors de la publication. Notons $R2$ celui retourné lors de la recherche. Maintenant, supposons que le protocole de gestion de vue n'ait pas encore atteint son objectif : les pairs rendez-vous ont des vues locales différentes, c'est-à-dire que la propriété (11.2) n'est pas satisfaite. Ils vont donc router les messages vers des destinations différentes, c'est-à-dire que $R1 \neq R2$. L'algorithme utilisé au sein de JXTA essaie de prendre en compte cette éventuelle différence. Il est notamment conçu pour que deux pairs ayant une vue locale similaire, mais pas forcément identique, désignent le même pair responsable ou un proche (dans l'espace de hachage) pour un même attribut indexé d'une annonce. Si, malgré tout, en cas de forte volatilité, l'algorithme de recherche désigne un mauvais pair responsable, la requête est transmise à son pair rendez-vous supérieur et inférieur dans l'espoir qu'il ait l'annonce de la ressource recherchée. La propagation de la requête se fait alors par parcours séquentiel de l'anneau des pairs rendez-vous, avec une durée de vie paramétrable pour pallier l'inconsistance des vues locales des pairs rendez-vous.

Ce mécanisme explique l'utilisation du terme *faiblement couplée* pour caractériser la table de hachage utilisée par JXTA pour la découverte de ressources. En effet, si des pairs rendez-vous apparaissent ou disparaissent, la correspondance des clés sur les pairs responsables n'est pas maintenue, contrairement aux DHT classiques. Mais elle le sera dans deux cas par les pairs standards : 1) périodiquement tous les intervalles de temps (paramètre DELTA_UPDATE_CYCLE, par défaut 30 secondes) et 2) lors de la connexion du pair qui a publié l'annonce sur un nouveau pair rendez-vous.

A.2.3 Algorithmique du service de canal virtuel

Le service de canaux virtuels utilise également le service SRDI pour la publication de tuples qui indiquent si un pair a créé un canal virtuel en entrée pour écouter les messages émis sur ce pipe. Les tuples émis sont stockés dans une table particulière du SRDI, JxtaPipeResolver, qui sera inspectée sur les pairs rendez-vous pour trouver les pairs qui écoutent sur le pipe. Lorsque le tuple du pipe est émis, une durée de vie maximale y est associée. Lors de la fermeture du canal virtuel en entrée (en anglais *input pipe*), le même tuple est réémis avec une durée de vie de zéro milliseconde. Ainsi, le pair de rendez-vous auquel est connecté le pair, mais également le pair responsable du pipe vont détruire le tuple. L'algorithme 3 présente alors le pseudo-code utilisé pour la résolution d'un pipe sur un pair rendez-vous. L'expression en entrée de la fonction de hachage est, de manière similaire au service de découverte, le type de l'annonce, l'attribut à chercher et sa valeur (ligne 1). Toutefois, avant de transmettre la requête au pair responsable de l'annonce, le SRDI local du pair

est vérifié et la requête peut être éventuellement transmise à un pair standard connecté à ce même pair rendez-vous (ligne 2 à 4). Comme pour la découverte de ressources (voir section A.2.2), et afin d'éviter de boucler sur le pair reponsable du pipe, un mécanisme de durée de vie pour la requête est utilisé. Ainsi, le calcul du pair responsable n'est effectué que sur le pair rendez-vous dont dépend le pair standard (lignes 6, 7 et 12). Si sur le pair responsable calculé, rien n'est trouvé dans le SRDI local, un parcours de l'anneau des pairs rendez-vous (walk) dans les deux sens est alors démarré (ligne 10).

Algorithme 3 : Pseudo-code utilisé sur un pair rendez-vous pour la résolution d'un canal virtuel unidirectionnel.

```
1  expression = Adv + Id + PipeID;
2  pair = vérifier SRDI local(expression);
3  si pair ! = null alors
4      transmettre(requête, pair);
5  sinon
6      si Nombre de sauts(requête) == 0 alors
7          responsable = getReplicaPeer(expression);
8      si responsable == null alors
9          Nombre de sauts(requête)++;
10         walk(query);
11     sinon
12         transmettre (requête, responsable);
```

La résolution d'un canal virtuel de diffusion fait intervenir un algorithme différent. En effet, compte tenu du mode de communication vers plusieurs pairs la recherche d'un unique pair qui a publié l'index d'un canal virtuel en entrée n'est pas suffisante. Ainsi, le calcul du pair responsable est inutile et la requête parcourt l'anneau des pairs rendez-vous dès son arrivée sur le pair rendez-vous dont dépend le pair standard (ligne 4). L'algorithme 4 présente le pseudo-code utilisé pour une telle résolution, sur un pair rendez-vous.

Algorithme 4 : Pseudo-code utilisé sur un pair rendez-vous pour la résolution d'un pipe de diffusion.

```
1  expression = Adv + Id + PipeID;
2  pairs = vérifier SRDI local(expression);
3  transmettre(requête, pairs);
4  walk(requête);
```

A.3 Mise en œuvre du modèle d'accès transparent aux données dans le modèle composant CCA

Dans cette annexe, nous présentons notre mise en œuvre de notre proposition pour l'intégration d'un modèle d'accès transparent aux données (voir chapitre 9) dans le modèle

composant CCA.

Afin de motiver cette présentation, nous avons choisi une application de lancer de rayon. Une telle application est utilisée pour calculer le rendu d'images de haute qualité. Elle suit le modèle de programmation maître-travailleurs pour distribuer des pixels formant une séquence d'images sur un ensemble de machines. Ce type d'application nécessite le partage de positions et de tampons d'images (en anglais *frame-buffer*). Le lecteur intéressé par une description détaillée du fonctionnement de ce type d'application peut se reporter à [30]. Une telle application peut se modéliser sous la forme d'un composant maître qui est en charge d'une collection de composants travailleurs. Ceux-ci accèdent en écriture aux tampons d'images de manière concurrente. Une application de lancer de rayon nécessite donc un partage transparent des différents tampons d'images. En outre, et contrairement à l'application décrite à la section 9.1.3, les données partagées sont passées en paramètre d'opérations de ports *provides* et de ports *uses*.

Le modèle composant CCA

CCA (*Common Component Architecture* [25]) est un ensemble de standards dont l'objectif est de développer une architecture commune pour la construction d'applications scientifiques haute performance, basées sur des composants logiciels largement testés. Un composant CCA définit de manière classique des ports *uses* et/ou *provides*. Ces ports sont similaires aux réceptacles et aux facettes d'un composant CCM. La spécification de ports CCA se fait par l'intermédiaire du langage de définition d'interfaces appelé *Scientific* IDL (SIDL). Toutefois et contrairement à CCM, CCA repose uniquement sur des appels à l'exécution pour la création de composants, l'ajout ou le retrait de ports. Ceci est réalisé via les différentes interfaces de programmation spécifiées par CCA.

Parmi l'ensemble de ces interfaces, trois jouent un rôle prépondérant. La première est l'interface Port qui est une interface vide à partir de laquelle tout port doit dériver. La deuxième est l'interface BuilderService qui permet la création et la composition de composants. Par exemple, un programmeur peut créer une instance de composant par l'intermédiaire de l'opération createInstance et connecter deux ports via l'opération connect. Cette interface permet également d'obtenir la liste des ports disponibles sur un composant par des mécanismes d'introspection. Enfin, la troisième interface s'appelle Services. Elle s'occupe de la gestion des ports au sein de la bibliothèque implémentant le modèle CCA. Une unique instance de cette interface est créée sur chaque instance de composant. Elle permet, par exemple, à un composant de déclarer un port *provides* et un port *uses* en appelant les opérations respectivement addProvidesPorts et registerUsesPort. Le composant peut alors obtenir une référence sur un port déclaré, par l'intermédiaire de l'opération getPort, avant d'invoquer une opération sur ce port.

Mise en œuvre du partage transparent de données : un exemple

Dans ce paragraphe nous décrivons comment les ports de données peuvent être spécifiés et implémentés dans CCA. Dans un premier temps, notre point de vue est celui du développeur de composants CCA. Dans un deuxième temps, nous prenons celui du développeur de la bibliothèque implémentant la spécification CCA.

```
interface ExtendedServices : Services {
void createAccessPort(in string  portName,
                      in string  typeDataName,
                      in TypeMap properties);
void createSharesPort(in string  portName,
                      in string  typeDataName,
                      in TypeMap properties);
};
```

```
interface AccessPort : Port {
  opaque get_pointer();
  long get_size();
  void acquire ();
  void release();
};
interface SharesPort : AccessPort {
  void associate(in opaque ptr,
                 in long size);
  void disassociate();
};
```

FIG. A.3 – Spécifications SIDL pour la projection dans le modèle CCA des ports de données.

Point de vue de l'utilisateur. Les ports de données sont distincts des ports *provides* et *uses*. En conséquence, comme l'illustre la partie gauche de la figure A.3, l'interface Service est étendue pour ajouter deux nouvelles opérations afin de gérer ces nouveaux type de ports. Les opérations createSharesPort et createAccessPort de cette nouvelle interface ExtendedService permettent de créer respectivement des ports *shares* et *accesses* du type spécifié.

Contrairement aux ports standards *provides* / *uses*, les ports de données définissent leur propre interface. Pour un port *accesses* (respectivement un port *shares*), un programmeur a accès à l'interface AccessPort (respectivement SharesPort). Le haut de la partie droite de la figure A.3 représente l'interface de programmation AccessPort. Cette interface contient des opérations pour obtenir un pointeur sur une donnée, écrire/lire et gérer la concurrence des accès. Le bas de la partie droite de la figure A.3 montre également l'interface SharesPort. Elle comprend les mêmes opérations que l'interface AccessPort, puisque cette interface hérite de AccessPort. Un programmeur peut en effet vouloir accéder à une donnée qu'il met à disposition. Toutefois, cette interface offre également deux nouvelles opérations pour gérer l'association de la donnée avec le port. L'opération associate fournit la capacité à un espace mémoire d'être attaché à un port de données, alors que disassociate correspond à l'opération opposée. Le développeur d'un composant a toujours la charge d'allouer et de libérer l'espace mémoire.

Le processus de connexion est également inchangé du point de vue du programmeur : la connexion entre un port *shares* et un port *accesses* est effectuée via l'opération connect. Toutefois, si les types des deux ports ne sont pas compatibles, une exception est levée.

Point de vue de l'implémenteur. Afin de fournir la vue décrite précédemment aux programmeurs de composants, l'implémentation de la spécification CCA doit, en interne, être en mesure de distinguer les ports classiques des ports de données. Cette distinction permet à la bibliothèque implémentant le modèle CCA d'effectuer les opérations appropriées. Par exemple, de créer et d'attacher une instance de l'interface AccessPort ou SharesPort quand un port de données est créé. Il faut également, lors de la connexion de deux ports, s'assurer de leur compatibilité en vérifiant leur type. Une autre modification de la bibliothèque implémentant le modèle CCA est nécessaire pour le processus de connexion à une interface AccessPort. Une référence à la donnée exportée par l'interface SharesPort doit être donnée à l'interface AccessPort. La référence correspond à l'identifiant de la donnée dans JUXMEM.

L'interface devant être indépendante du type de la donnée, nous avons choisi d'utiliser le type opaque SIDL afin de représenter un pointeur sur la mémoire locale.

Pour résumer, notre proposition d'extension des spécifications CCA pour le support des ports de données nécessite des modifications de l'interface Services et une réimplémentation de l'interface BuilderService. Les opérations connect, getPort ainsi que celles permettant l'introspection sont également concernées. Toutefois, les modifications à apporter sont minimes et très ciblées.

Pour autoriser le partage de données lors d'appels d'opérations dans CCA, notre mise en œuvre se base sur les ports de données. La nouvelle notation introduite (« & ») dans le modèle abstrait étant identique à celle utilisée dans les interfaces des ports de données, la projection est directe. Du point de vue du programmeur d'une bibliothèque de composants, les seules modifications se situent dans le langage SIDL. Il doit être enrichi de notre notation « & ». L'implémentation de cette notation est générée par les compilateurs SIDL, dans les souches (en anglais *stubs*) et squelettes (en anglais *skeleton*) des ports *provides* et *uses*. Elle repose sur nos ports de données, tels que la figure A.3 les définit. Il s'agit de convertir automatiquement les paramètres partagés, du côté du composant fournissant l'opération, en ports *accesses* ou *shares* selon le mode du paramètre. Les compilateurs SIDL tel que Babel [112] doivent donc être modifiés.

Évaluation

Contrairement à la mise en œuvre sur le modèle CCM, celle sur le modèle CCA n'a pas été effectuée. Par ailleurs, aucune évaluation pratique n'a été réalisée. Toutefois, cela pourrait faire l'objet de travaux futurs. Cependant, si nous reprenons l'exemple d'une application de lancer de rayon, nos propositions d'extension des modèles composant pour un partage transparent des données permet d'en simplifier le développement. En effet et comme nous l'avons vu au début de l'annexe A.3, ce type d'application suit un modèle maître-travailleurs pour le calcul de rendu d'images de haute qualité. Or, le tampon de données doit par exemple être accédé en écriture par les travailleurs lors de l'appel de leur fonction de calcul de rendu. L'utilisation de notre proposition de ports de données pour le passage en paramètres de données partagées (voir section 9.2.2) permet de réaliser cela de manière transparente. Par ailleurs, ce passage par référence via notre notation « & » permet théoriquement d'améliorer les performances d'accès aux données par les travailleurs (voir section 8.4). En effet et de manière similaire à ce qui est réalisé dans l'utilisation de JUXMEM par DIET, les données ne sont plus copiées à chaque appel de la fonction de calcul de rendu d'un travailleur. Elles sont accédées à la demande, en bénéficiant généralement des performances d'accès d'un réseau local et non à longue distance.

A.4 Définition du langage de description JDL

La figure A.4 présente la définition (DTD pour Déclaration de Type de Document) du langage de description JDL d'application basées sur JXTA.

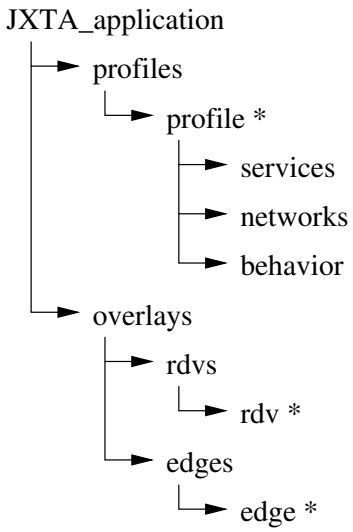

FIG. A.4 – Définition du langage de description JDL d'applications basées sur JXTA.

Mathieu JAN

JUXMEM : un service de partage transparent de données pour grilles de calcul fondé sur une approche pair-à-pair.

Mots-clés : grille de calcul, service de partage de données, pair-à-pair, intergiciels, JUXMEM, JXTA, communications haute performance.

La quantité importante de ressources offertes par les grilles de calcul permet d'envisager la résolution de problèmes de plus en plus complexes. La programmation simple et efficace de tels environnements est un véritable défi. La plupart des travaux réalisés dans ce domaine ont visé à élaborer des modèles de programmation offrant une gestion efficace et transparente de la puissance de calcul offerte par les grilles de calcul. Toutefois, la gestion des données dans de telles infrastructures est comparativement rudimentaire. Elle reste à la charge des développeurs d'applications sur de nombreux aspects : localisation des données, cohérence, tolérance aux fautes, etc.

La contribution de ce travail de doctorat est de spécifier le concept de *service de partage de données* pour grilles, afin d'intégrer aux modèles de programmation actuels un *modèle d'accès transparent aux données*. Notre proposition s'inspire essentiellement des systèmes à mémoire virtuellement partagée (MVP) et des systèmes pair-à-pair (P2P). Cette approche hybride a pour objectif de conserver les points forts des systèmes à MVP (transparence et gestion de la cohérence) grâce à une organisation fondée sur les points forts des systèmes P2P (passage à l'échelle et tolérance aux fautes). Afin de valider notre concept de service de partage de données, nous proposons une architecture appelée JUXMEM (pour *Juxtaposted Memory*) ainsi qu'une implémentation s'appuyant sur la spécification P2P JXTA. Cette mise en œuvre a été validée par son intégration dans deux modèles de programmation, utilisés pour concevoir les applications s'exécutant sur les grilles de calcul : le modèle Grid-RPC et les modèles à base de composants logiciels. Les bénéfices de notre approche ont été évalués à grande échelle sur la grille expérimentale Grid'5000.

Ce manuscrit décrit également deux autres contributions qui se situent dans le contexte de l'utilisation de techniques P2P, pour la construction d'intergiciels destinés aux grilles de calcul. Au-delà de notre objectif premier, qui a été d'adapter JXTA pour permettre une mise en œuvre efficace de notre service, ces contributions ont une portée plus générale. Elles concernent d'une part l'utilisation de la plate-forme P2P JXTA sur les grilles de calcul et d'autre part à l'optimisation de ses couches de communication dans ce contexte particulier d'exécution.