Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul

Alexandre Denis

▶ To cite this version:

Alexandre Denis. Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul. Réseaux et télécommunications [cs.NI]. Université Rennes 1, 2003. Français. NNT : . tel-00009595

HAL Id: tel-00009595
https://theses.hal.science/tel-00009595v1

Submitted on 24 Jun 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Numéro d'ordre de la thèse : 2964

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

pour obtenir

le grade de *DOCTEUR DE L'UNIVERSITÉ DE RENNES 1*

Mention INFORMATIQUE

PAR

Monsieur Alexandre DENIS

Équipe d'accueil : projet PARIS, IRISA, Rennes

École Doctorale MATISSE

Composante universitaire : IFSIC/IRISA

Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul

SOUTENUE LE 12 décembre 2003 devant la Commision d'Examen

COMPOSITION DU JURY

Monsieur Henri BAL, Professeur	membre du jury
Monsieur Michel COSNARD, Professeur	membre du jury
Monsieur Bertil FOLLIOT, Professeur	membre du jury et rapporteur
Monsieur Jean-Marc GEIB, Professeur	membre du jury et rapporteur
Monsieur Christian PÉREZ, Chargé de Recherche	membre du jury et co-encadreur
Monsieur Thierry PRIOL, Directeur de Recherche	membre du jury et directeur de thèse

Remerciements

Jusqu'à présent, la première chose — et parfois la seule ! — que je lisais dans un manuscrit de thèse était la page « remerciements ». C'est maintenant déjà mon tour ; je n'ai pas vu le temps passer durant ces trois années de thèse ! Je tiens à remercier ici tous ceux qui m'ont aidé, soutenu et encouragé pendant ma thèse.

Mes premiers remerciements vont bien entendu à mon jury, sans qui ce document ne serait qu'un morceau de papier sans grande valeur. Je tiens tout d'abord à remercier Michel Cosnard pour m'avoir fait l'honneur de présider mon jury, alors même qu'il était en passe d'être nommé à un poste qui ne lui laisse que peu de temps libre. Je remercie également chaleureusement Bertil Folliot et Jean-Marc Geib, tous deux rapporteurs, qui ont consacré une partie de leur temps précieux à relire ce manuscrit et à faire des commentaires constructifs. Je remercie Henri Bal qui, bien que n'ayant pas pu faire le déplacement le jour de la soutenance, a fait preuve d'un enthousiasme qui fait chaud au cœur face à mes travaux. Et évidemment, n'oublions pas mes deux encadrants, Thierry Priol et Christian Pérez, qui m'ont fait confiance pendant ces trois ans, et qui m'ont donné l'impression de savoir où j'allais y compris quand moi-même je ne le savais pas !

Tout a commencé alors que je n'avais pas dix ans, que ni les grilles ni internet n'existaient, quand ma maman a ramené à la maison un objet étrange, un Tandy Radio Shack TRS80. J'ai attrapé le virus. Depuis, je n'ai pas décroché ! J'ai fait mon chemin, en croisant au passage M. Juhel, mon professeur d'informatique en prépa au Lycée Faidherbe, qui m'a encouragé à continuer dans cette voie sérieusement et qui a cru en moi ; je l'en remercie chaleureusement. La concrétisation est venue avec mon entrée à l'ENS Lyon. Que tous ceux qui ont contribué à l'instauration — providentielle puisque je fait partie de la première promotion — du concours de la série informatique *à l'ENS soient remerciés pour avoir donné ses lettres de noblesse à l'informatique.*

Mais la révélation qui m'a définitivement converti au système et aux réseaux, je la dois aux cours de Raymond Namyst. Il a réussi à faire aimer ce domaine à tous les étudiants qui assistaient à ses cours, et encore plus à ceux qui, comme moi, avaient le goût de « comprendre comment ça marche ». Ma route a ensuite croisé celle de Luc Bougé, à qui je dois le goût du travail bien fait, le départ pour Rennes, et la rencontre de Thierry Priol. Un grand merci à tous ces gens sans qui cette thèse n'aurait sans doute pas eu lieu.

Nous en arrivons maintenant à Rennes et à l'IRISA. Je tiens à remercier tous le projet PARIS, Christian et André pour toutes les discussions mémorables dans le bureau, nos voisins successifs pour avoir supporté ces discussions et ma voix qui, selon la réputation largement surfaite, porte à l'autre bout du couloir, Yvon Jégou pour ses interventions précises, rapides et efficaces sur la grappe, Maryse et Élodie toujours disponibles, efficaces et de bonne humeur. Merci à Vincent Danjean, Raymond Namyst et Olivier Aumage pour avoir fait PM^2, pour le faire évoluer selon mes besoins, et pour tout le reste ; merci à Guillaume Mercier pour MPICH/Mad et pour m'avoir ouvert à l'esthétique de Steven Seagal. Merci aussi à tous les autres que j'oublie ici et qui ont contribué d'une façon ou d'une autre à cette thèse.

Enfin, je tiens à remercier mes parents qui n'avaient pas réalisé tout de suite qu'écrire une thèse est un travail titanesque mais s'en sont rendu compte depuis, et Chantal qui, elle, s'en est aperçue très vite quand, honteusement, je passais plus de temps avec mon manuscrit qu'avec elle !

Table des matières

Table des figures ix

Liste des définitions xi

1 Introduction 1

Partie I – **Contexte d'étude** 5

2 Les systèmes concurrents 7
- 2.1 Définitions et paramètres d'étude . . . 8
- 2.2 Les systèmes parallèles . . . 15
- 2.3 Les systèmes répartis . . . 22
- 2.4 Conclusion . . . 31

3 Les grilles de calcul 33
- 3.1 Notion de grille de calcul . . . 33
- 3.2 Voir une grille comme un calculateur parallèle . . . 37
- 3.3 Voir une grille comme un système réparti . . . 38
- 3.4 Nouvelles possibilités des grilles . . . 40
- 3.5 Conclusion . . . 42

4 Analyse des exécutifs communicants 43
- 4.1 Analyse de l'abstraction des ressources : portabilité et généricité . . . 44
- 4.2 Offrir plusieurs paradigmes simultanément . . . 47
- 4.3 Conclusion . . . 56

Partie II – **Un modèle de plate-forme de communication pour les grilles** 57

5 Un modèle de plate-forme d'intégration d'exécutifs communicants 59
- 5.1 Vers un modèle d'abstraction multi-paradigme . . . 60
- 5.2 Architecture pour une plate-forme de communication multi-paradigme pour les grilles 67
- 5.3 Intégrer différents modèles d'exécution . . . 71
- 5.4 Conclusion . . . 74

6 Des abstractions de communication multi-paradigme 75
- 6.1 Principe de la couche d'adaptation d'abstraction . . . 75
- 6.2 L'abstraction du réparti . . . 78
- 6.3 L'abstraction du parallélisme à mémoire distribuée . . . 82

6.4 L'abstraction du parallélisme à mémoire partagée . . . 85
6.5 Généralisation des mécanismes d'adaptation d'abstraction . . . 86
6.6 Sélection automatique et assemblage des adaptateurs . . . 90
6.7 Conclusion . . . 92

7 Virtualisation des interfaces de communication **93**
7.1 Contexte d'intégration . . . 93
7.2 Principe de la virtualisation des ressources . . . 94
7.3 Virtualisation par personnalités . . . 96
7.4 Transparence . . . 98
7.5 Discussion sur la virtualisation . . . 101
7.6 Conclusion . . . 101

8 Accès arbitré aux ressources **103**
8.1 Méthodologie d'accès arbitré . . . 104
8.2 Mise en commun de la scrutation . . . 107
8.3 Mécanismes de multiplexage . . . 110
8.4 Compétition et équité . . . 111
8.5 Harmonisation des actions globales . . . 115
8.6 Discussion et conclusion . . . 116

Partie III – **Mise en œuvre et évaluation** **117**

9 La plate-forme PadicoTM **119**
9.1 Présentation de PadicoTM . . . 120
9.2 Fondations de PadicoTM . . . 121
9.3 Le niveau abstrait dans PadicoTM . . . 130
9.4 Adaptateurs d'abstraction . . . 140
9.5 Personnalités et exécutifs sur PadicoTM . . . 142
9.6 Conclusion . . . 144

10 Évaluation de PadicoTM **145**
10.1 Micro-benchmark de *NetAccess* . . . 145
10.2 Évaluation des abstractions *VLink* et *Circuit* . . . 150
10.3 Performances des exécutifs sur PadicoTM . . . 150
10.4 Méthodes de communication pour WAN . . . 155
10.5 Conclusion . . . 157

11 Utilisations de PadicoTM **159**
11.1 CORBA parallèle : les environnements PaCO++ et GRIDCCM . . . 159
11.2 Projet GRID-RMI . . . 160
11.3 Projet HydroGRID . . . 160
11.4 Projet EPSN . . . 161
11.5 L'environnement DIET et le projet GASP . . . 162
11.6 Projet ALTA . . . 162
11.7 Conclusion . . . 163

12 Conclusion et perspectives **165**

Bibliographie **171**

Table des figures

2.1 Un intergiciel pour cacher la répartition géographique des ressources. . . . 12
2.2 Implémentations d'exécutifs liées au réseau. . . . 13
2.3 Portabilité assurée par des pilotes. . . . 13
2.4 Exécutifs sur un environnement générique de communication 14

3.1 GRIDCCM : des composants CORBA parallèles utilisent CORBA et MPI. . . . 41

5.1 Utiliser une interface abstraite parallèle en dénominateur commun. . . . 61
5.2 Utiliser une interface abstraite répartie en dénominateur commun. . . . 63
5.3 Utiliser une interface abstraite unifiée. . . . 64
5.4 Utiliser différentes interfaces abstraites pour différents paradigmes 66
5.5 Modèle proposé avec les paradigmes parallèle à mémoire distribuée et réparti. . . . 67
5.6 Vue générale de l'architecture de plate-forme de communication proposée. . . . 69

6.1 Représentation d'une collection principale d'adaptateurs pour trois paradigmes. . . . 78
6.2 Exemple de décomposition en sous-circuits. . . . 83
6.3 Exemple d'adaptateurs alternatifs 87
6.4 Exemple d'intercepteur 87
6.5 Adaptateurs croisés abstraits 88
6.6 Adaptateur croisé abstrait et adaptateur croisé 88

7.1 Principe de la virtualisation. . . . 95
7.2 Connexions virtuelles entre les couches logicielles. . . . 95
7.3 La virtualisation permet une portabilité transparente. . . . 96
7.4 Virtualisation par personnalités 98
7.5 Transparence : l'ensemble personnalité + abstraction se comporte comme l'implémentation réelle. . . . 99

8.1 Arbitrer l'accès aux ressources 105
8.2 Latence de transmission mesurée sur les SAN courants. . . . 111

9.1 Exemple de description de module : le module "Console". . . . 121
9.2 Exemple de description de module : le module "CORBA-omniORB-4.0". . . . 122
9.3 L'interface de *Puk*. . . . 122
9.4 Débit de transmission sur Myrinet-2000 en fonction de la stratégie d'allocation de la mémoire en réception 123
9.5 L'interface de *SysIO*. . . . 126
9.6 L'interface de *MadIO*. . . . 128
9.7 Agrégation d'en-tête 128
9.8 L'interface abstraite *VLink*. . . . 131
9.9 Exemples d'utilisation de l'observateur 131
9.10 Exemple d'utilisation des primitives *VLink* de gestion de lien 132

9.11 Exemple d'utilisation des primitives *VLink* d'échange de données 133
9.12 L'interface abstraite *Circuit*. 134
9.13 Exemple d'échange d'un message de longueur arbitraire avec l'interface *Circuit*. 134
9.14 Exemple de topologie d'étude 135
9.15 Structures PadicoTM correspondant à la topologie de la figure 9.14 du point de vue du nœud *node0*. 136
9.16 Structures de description des adaptateurs. 136
9.17 Représentation de la partie de la collection principale d'adaptateurs offrant l'interface *VLink*. 137
9.18 Exemple de déroulement de sélection automatique : sélection de *MadIO*. 138
9.19 Exemple de déroulement de sélection automatique : interopérabilité avec TCP/IP. . . . 139
9.20 Exemple de déroulement de sélection automatique : utilisation d'un adaptateur alternatif. 139

10.1 Comparaison de *MadIO* et Madeleine sur Myrinet-2000. 146
10.2 Comparaison de *SysIO* et *sockets* sur Ethernet-100. 148
10.3 Latence et débit mesurés sur *MadIO* avec deux clients simultanés. 149
10.4 Programme de test utilisé pour mesurer la concurrence entre scrutation et calculs. . . . 149
10.5 Latence et débit des abstractions *Circuit* et *VLink* sur *MadIO*/Myrinet-2000. 150
10.6 Latence et débit de MPICH-padico et MPICH/*Madeleine* sur Myrinet-2000. 151
10.7 Latence et débit de diverses implémentations CORBA sur Myrinet-2000. 152
10.8 Comparaison des performances de divers exécutifs et interfaces sur PadicoTM. 154
10.9 Coût logiciel et matériel estimé de chaque module lors de l'envoi de messages. 155
10.10 Débit mesuré sur VTHD avec des flux parallèles. 156

11.1 Exemple d'utilisation de GRIDCCM. 160
11.2 Architecture logicielle pour le projet GRID-RMI. 161

Liste des définitions

2.1 Fil d’exécution (tâche) 8
2.2 Systèmes concurrents 8
2.3 Système distribué 8
2.4 Paradigme 8
2.5 Paradigme de programmation (modèle de programmation) 8
2.6 Paradigme de communication 8
2.7 Intergiciel (logiciel médiateur, *middleware*) 11
2.8 Exécutif (support exécutif, *runtime*) 11
2.9 Environnement, plate-forme de communications 12
2.10 Exécutif communicant 12
2.11 Portabilité 13
2.12 Générique, spécifique 14
2.13 Composant 15
2.14 Calcul parallèle 15
2.15 Calcul parallèle à mémoire distribué 16
2.16 Calcul parallèle à mémoire partagée 16
2.17 Système réparti 23
3.1 Grille 34
5.1 Interface de communication 60
5.2 Abstraction de communication 60
7.1 Virtualisation 93

Chapitre 1

Introduction

LE CALCUL SCIENTIFIQUE est, depuis les origines de l'informatique, un moteur pour la recherche de solutions toujours plus performantes, plus robustes, et plus simples à utiliser. Des premières applications militaires de déchiffrement de codes secrets et de simulations nucléaires, jusqu'aux applications récentes en biologie — avec des initiatives médiatiques telle le décrypthon —, les applications gourmandes en puissance de calcul ne manquent pas. La puissance des machines est en constante progression depuis le début ; le secteur affiche une croissance soutenue. Des solutions originales ont été trouvées pour améliorer les processeurs ; le parallélisme a permis de dépasser la croissance des processeurs seuls en agrégeant la puissance de plusieurs processeurs. Ce n'était pas encore suffisant : des applications demandaient une puissance de calcul encore supérieure.

Dans le même temps, les réseaux progressaient régulièrement, de façon indépendante. Au début des années 1960, Joseph C.-R. Licklider, alors directeur du département *Command and Control Research* de l'ARPA, imagina un réseau qui allait devenir *arpanet* puis *internet*. Il avait remarqué le premier l'esprit de communauté que le partage d'une machine entraînait entre les utilisateurs. Il pressentait déjà que « *la promesse des ordinateurs en tant que médium de communication entre les gens rendrait insignifiantes les origines historiques des ordinateurs en tant que machines de calcul* ». Dans les années 1995-2000, *Internet* s'est répandu et démocratisé à une vitesse fulgurante, dépassant probablement tout ce que Licklider avait pu imaginer. Devenu un moyen de communication largement répandu, fiable et rapide, *internet* avait réellement atteint un niveau de maturité suffisant pour être considéré par le grand public comme « un médium de communication entre les gens ».

C'est cette réalisation de la prophétie de Licklider au-delà de toute espérance qui allait susciter soudainement l'intérêt de la communauté du calcul scientifique pour le réseau mondial. Les réseaux étant devenus suffisamment rapides, ils devenaient utilisables pour relier des machines de calcul à grande échelle et construire ce qui allait s'appeler les *grilles de calcul*. On rentrait alors de plein pieds dans l'ère suivante après le parallélisme.

Cependant, la façon de programmer les grilles de calcul n'a pas suivi l'évolution au même rythme. On a d'abord voulu les programmer simplement comme des machines parallèles, au mépris de leurs spécificités. Cette façon de faire bridait leurs possibilités. Il a alors été proposé un modèle de programmation qui réconcilie l'approche parallèle, familière de la communauté du calcul, et l'approche répartie, familière de la communauté des communications. Toutefois, ce modèle de programmation ne peut s'exprimer pleinement sans des infrastructures logicielles capables de tirer profit des ressources de communication des grilles.

Objectif

Nous étudions dans ce document la rencontre entre le calcul parallèle et les systèmes répartis sur les grilles de calcul, du point de vue de la gestion des communications. Il s'agit de pouvoir utiliser les différentes ressources des grilles au mieux, sans contraindre la façon dont les applications les utilisent ; en particulier, il est souhaitable qu'une application puisse voir une grille comme un système réparti, comme un système parallèle, ou comme une combinaison, indépendamment de ce que les ressource sont réellement.

Nous proposons une approche basée sur une plate-forme de communication qui tient compte des spécificité des grilles. Cette plate-forme est capable de proposer une vision virtuelle des grilles de calcul : les applications peuvent choisir la vision qui convient le mieux au modèle de programmation qu'elles emploient — parallèle, réparti, ou une combinaison — indépendamment des ressources physiques réellement utilisées. La plate-forme de communication doit également être capable d'utiliser chaque ressource d'une grille au mieux, en accord avec sa nature : les machines parallèles qui la composent selon les méthodes du parallélisme, et les réseaux de communication qui les relient selon les méthodes du réparti. Cette plate-forme est à même de proposer la vision choisie par l'application sur les ressources disponibles, en utilisant au mieux les ressources avec les méthodes appropriées.

Dans ce document, nous proposons une architecture pour une telle plate-forme de communication basée sur la dualité parallèle-répartie des grilles de calcul. Cette architecture est modulaire pour permettre une adaptation aux ressources variées des grilles, et aux besoins variés des applications. La modularité permet en outre une extensibilité par ajout de modules qui apportent de nouveaux protocoles, méthodes de communication, ou réseaux supportés. De plus, nous avons porté une attention particulière à l'interopérabilité avec les exécutifs standard qui n'utilisent pas notre architecture de plate-forme de communication. Cette architecture est organisée selon trois grandes directions :

- un *arbitrage* d'accès aux ressource permet à plusieurs exécutifs d'accéder simultanément et équitablement aux ressources de communication. Cet arbitrage est essentiellement basé sur une mutualisation des mécanismes de scrutation, et sur une régulation pour tenir compte des autres tâche concurrentes ;
- une *adaptation d'abstraction* permet d'abstraire la vision des ressources de communication. L'adaptation d'abstraction permet par exemple d'utiliser des réseaux haute performance conçus pour le parallélisme avec une interface ou un exécutif de type réparti, ou encore de déployer le communicateur d'un exécutif parallèle sur un ensemble de nœuds qui recouvre plusieurs machines parallèles voire plusieurs site ;
- une *virtualisation* des interfaces permet de faire utiliser les mécanismes d'adaptation d'abstraction par des exécutifs et applications sans les modifier. Ceci autorise la réutilisation de codes existants, sans devoir refaire de lourds efforts de développement ; ceci autorise également le développement de nouveaux codes qui utilisent la plate-forme sans lier ces codes à la plate-forme.

Nous décrivons dans ce document notre architecture de plate-forme de communication pour les grilles de calcul, et nous présentons et évaluons une mise en œuvre de ces principes dans la plate-forme PadicoTM. La plate-forme PadicoTM a été rendue publique sous licence GNU et enregistrée à l'Agence pour la Protection des Programmes (APP). Elle a été intégrée en tant que contribution au consortium *Gelato* [112]. PadicoTM est utilisé par divers projets, en particulier PaCO++/GRIDCCM, DIET [56], EPSN, et HydroGRID.

Organisation de ce document

Ce document s'organise en trois parties. La première partie présente le contexte de notre étude sous la forme de trois chapitres : une présentation générale des systèmes concurrents et de leurs problématiques, une étude plus spécialement sur les grilles de calcul, et enfin une analyse de l'adéquation des exécutifs aux besoins en matière de communication sur les grilles de calcul.

La deuxième partie décrit notre modèle de plate-forme de communication pour les grilles, dont la présentation est décomposée en quatre chapitre. Nous procédons d'abord à une étude et présentation

générale de l'architecture globale d'une telle plate-forme de communication, puis nous détaillons chacun des trois niveaux de la plate-forme : un modèle d'abstraction multi-paradigme, la virtualisation des interfaces de communication, et un accès arbitré au ressources.

La troisième partie étudie une mise en œuvre de ce modèle de plate-forme de communication. Nous présentons d'abord la mise en œuvre dans la plate-forme PadicoTM, puis nous évaluons ses performances, et enfin nous décrivons son utilisation dans plusieurs projets.

Dans le dernier chapitre nous tirons les conclusions de notre étude et traçons un aperçu des travaux futurs qui s'inscrivent dans la perspective de nos travaux.

Publications

Les travaux que nous présentons dans ce document ont engendré des publications dans diverses conférences et revues. Ces publications concernent :

- l'architecture de plate-forme de communication et sa mise en œuvre dans PadicoTM [5, 10, 11, 16, 17] ;
- l'approche proposée par Padico [12, 2] et le concept d'objet CORBA parallèle [4, 9] ;
- une étude ciblée spécifiquement sur les relations entre CORBA et les réseaux haute performance [6, 15] ;
- les interactions entre la plate-forme PadicoTM et Madeleine [3, 1, 8].

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

Chapitres de livres

[1] Olivier AUMAGE, Luc BOUGÉ, Alexandre DENIS, Lionel EYRAUD, Raymond NAMYST, et Christian PÉREZ. « *Calcul réparti à grande échelle. Métacomputing* », Chapitre Communications efficaces au sein d'une interconnexion hétérogène de grappes, pages 103–128. Hermès/Lavoisier, Mai 2002.

[2] Alexandre DENIS, Christian PÉREZ, Thierry PRIOL, et André RIBES. « *Process Coordination and Ubiquitous Computing* », Chapitre Programming the Grid with Distributed Objects, pages 133–148. CRC Press, 2003.

Articles dans des revues internationales

[3] Olivier AUMAGE, Luc BOUGÉ, Alexandre DENIS, Lionel EYRAUD, Jean-François MÉHAUT, Guillaume MERCIER, Raymond NAMYST, et Loïc PRYLLI. « High Performance Computing on Heterogeneous Clusters with the Madeleine II Communication Library ». *Cluster Computing*, 5:43–54, 2002.

[4] Alexandre DENIS, Christian PÉREZ, et Thierry PRIOL. « Achieving Portable and Efficient Parallel CORBA Objects ». *Concurrency and Computation: Practice and Experience*, 15(10):891–909, Août 2003.

[5] Alexandre DENIS, Christian PÉREZ, et Thierry PRIOL. « PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes ». *Future Generation Computer Systems*, 19:575–585, 2003.

Articles dans des revues nationales

[6] Alexandre DENIS. « CORBA haute performance ». *Technique et Science Informatiques (TSI)*, 21:659–683, 2002.

Articles dans des revues en ligne

[7] Alexandre DENIS, Christian PÉREZ, Thierry PRIOL, , et André RIBES. « Parallel CORBA Objects for Programming Computational Grids ». *Distributed Systems Online*, 4(2), Février 2003. Electronic journal (`http://dsonline.computer.org/0302/f/pri_print.htm`).

Conférences internationales avec comité de lecture

[8] Olivier AUMAGE, Luc BOUGÉ, Alexandre DENIS, Jean-François MÉHAUT, Guillaume MERCIER, Raymond NAMYST, et Loïc PRYLLI. « A Portable and Efficient Communication Library for High-Performance Cluster Computing ». Dans *IEEE Intl Conf. on Cluster Computing (Cluster 2000)*, pages 78–87, Technische Universität Chemnitz, Allemagne, Novembre 2000.

[9] Alexandre DENIS, Christian PÉREZ, et Thierry PRIOL. « Portable parallel CORBA objects: an approach to combine parallel and distributed programming for Grid Computing ». Dans *Proc. of the 7th Intl. Euro-Par'01 Conference (EuroPar'01)*, pages 835–844, Manchester, UK, Août 2001. Springer. Article sélectionné pour publication en tant que [4].

[10] Alexandre DENIS, Christian PÉREZ, et Thierry PRIOL. « Towards High Performance CORBA and MPI Middlewares for Grid Computing ». Dans Craig A. LEE, éditeur, *Proc of the 2nd International Workshop on Grid Computing*, numéro 2242 dans LNCS, pages 14–25, Denver, Colorado, USA, Novembre 2001. Springer-Verlag. In conjunction with SuperComputing 2001 (SC'01).

[11] Alexandre DENIS, Christian PÉREZ, et Thierry PRIOL. « PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes ». Dans *IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 144–151, Berlin, Allemagne, Mai 2002. IEEE Computer Society. Sélectionné pour publication en tant que [5].

[12] Alexandre DENIS, Christian PÉREZ, Thierry PRIOL, et André RIBES. « Padico: A Component-Based Software Infrastructure for Grid Computing ». Dans *17th International Parallel and Distributed Processing Symposium (IPDPS2003)*, Nice, France, Avril 2003. IEEE Computer Society.

Conférences nationales avec comité de lecture

[13] Alexandre DENIS. « Adaptation de l'environnement générique de métacomputing Globus à des réseaux hauts débits ». Dans *Actes des Journées Doctorales Informatique et Réseaux (JDIR 2000)*, pages 121–130, Univ. Paris 6, Novembre 2000.

[14] Alexandre DENIS. « VRP : un protocole avec une tolérance de perte ajustable pour des hautes performances sur réseau longue distance ». Dans *Actes des Rencontres francophones du parallélisme (RenPar 12)*, pages 27–32, LIB, Univ. Besançon, Juin 2000. Version abrégée de [19].

[15] Alexandre DENIS. « CORBA et réseaux haute performance ». Dans *13èmes Rencontres Francophones du Parallélisme (RenPar'13)*, pages 189–194, Paris, Avril 2001.

[16] Alexandre DENIS. « PadicoTM : un environnement ouvert pour l'intégration d'exécutifs communiquants ». Dans *14èmes Rencontres Francophones du Parallélisme (RenPar'14)*, pages 99–106, Hammamet, Tunisie, Avril 2002.

[17] Alexandre DENIS. « Architecture d'une plate-forme de communication multi-paradigme pour les grilles ». Dans *15èmes Rencontres Francophones du Parallélisme (RenPar'15)*, La Colle sur Loup, Octobre 2003. À paraître.

Actes d'écoles sans comité de lecture

[18] Alexandre DENIS, Christian PÉREZ, Thierry PRIOL, et André RIBES. « Chapitre 2 : Composants logiciels et grilles de calcul ». Dans Emmanuel JEANNOT, éditeur, *École thématique sur la globalisation des ressources informatiques et des données*, pages 15–27, Aussois, France, Décembre 2002. INRIA.

Divers

[19] Alexandre DENIS. « Variable Reliability Protocol: A protocol with a tunable loss tolerance for high performance over a WAN ». Rapport Technique RR2000-11, LIP, 2000. `http://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2000/RR2000-11.ps.Z`.

[20] Alexandre DENIS et Christian PÉREZ. « *Padico user's manual* ». IRISA. `http://www.irisa.fr/paris/Padico/`.

Première partie

Contexte d'étude

Chapitre 2

Les systèmes concurrents

Sommaire

2.1 Définitions et paramètres d'étude 8
2.1.1 Terminologie des systèmes concurrents 8
2.1.2 Problématiques des systèmes concurrents 9
2.1.3 Infrastructures logicielles 11
2.2 Les systèmes parallèles 15
2.2.1 Architectures parallèles 16
2.2.2 Modèles de programmation parallèle 17
2.2.3 Infrastructures logicielles pour le parallélisme 18
2.2.4 Conclusion sur les systèmes parallèles 22
2.3 Les systèmes répartis 22
2.3.1 Architectures réparties 23
2.3.2 Modèles de programmation répartie 24
2.3.3 Infrastructures logicielles pour le calcul réparti 24
2.3.4 Conclusion des systèmes répartis 30
2.4 Conclusion 31

Dans ce chapitre, nous effectuons un tour d'horizon du contexte général de notre étude : les systèmes concurrents. La notion de systèmes concurrents est apparue avec les machines reliées par des réseaux et les machines à plusieurs processeurs ; dans les deux cas, plusieurs "activités" coordonnées ont lieu simultanément pour un but commun. La diffusion de plus en plus large des techniques du parallélisme, en particulier les grappes de PC, la récente explosion du développement d'*internet*, et la généralisation des opérations réalisées par l'intermédiaire des réseaux ont largement accru l'intérêt pour les systèmes concurrents.

Ce chapitre est subdivisé en trois parties. Nous présentons tout d'abord les idées générales communes des systèmes concurrents, à savoir les définitions fondatrices, les problématiques, et l'organisation des infrastructures logicielles. Nous nous focalisons ensuite sur deux cas particuliers de systèmes concurrents : les systèmes parallèles et les systèmes répartis. Nous étudierons successivement les deux paradigmes selon la même démarche, à savoir : les architectures, les modèles de programmation, les infrastructures logicielles utilisées pour les mettre en œuvre. Enfin, nous conclurons par une comparaison des deux approches qui se sont développées indépendamment l'une de l'autre. Elles ont de nombreuses problématiques communes, beaucoup de points communs dans leurs solutions, mais aboutissent pourtant à des résultats très différents, fortement teintés des priorités qui ont été attribuées aux différentes problématiques dans chaque paradigme. Finalement, nous traçons les premières lignes de la convergence entre systèmes répartis et systèmes parallèles, convergence qui annonce les grilles de calcul qui font l'objet du chapitre suivant.

2.1 Définitions et paramètres d'étude

Nous commençons ce chapitre par les définitions générales des notions que nous manipulerons dans la suite de ce document. Cette section s'intéresse d'abord aux systèmes concurrents en général, puis aux problématiques amenées par les systèmes concurrents, et enfin étudie les infrastructures logicielles pour les programmer. Nous faisons référence à ces définitions tout au long du document. La liste de toutes les définitions posées dans ce document est donnée à la page *xi*.

2.1.1 Terminologie des systèmes concurrents

Dans cette section, nous définissons les notions essentielles des systèmes concurrents. La caractéristique d'un système concurrent est qu'un ensemble d'actions se déroulant en même temps (sur des processeurs, machines, ou sites différents) sont liées entre elles au niveau logique en un seul système coordonné. Pour définir précisément de tels systèmes, nous définissons d'abord la notion de *fil d'exécution*.

Définition 2.1 : ***fil d'exécution (tâche)*** **—** Un fil d'exécution est une suite logique d'actions résultant de l'exécution d'un programme. Un programme peut être constitué de plusieurs fils d'exécution ; on parle alors dans certains cas de *multiprogrammation* ou de *multi-threading*.

Définition 2.2 : ***systèmes concurrents*** **—** Nous définissons un système concurrent comme un ensemble (pouvant varier dans le temps) de fils d'exécution qui interagissent pour parvenir à un but commun.

Notons qu'un système concurrent peut contenir plusieurs programmes. Suivant ces définitions, des exemples usuels de systèmes concurrents sont les machines multi-processeur et/ou parallèles, les grappes de PC, ou encore le système constitué d'un serveur *web* et d'un ou plusieurs clients.

Pour une caractérisation plus fine des systèmes concurrents, nous ajoutons le critère de la distribution géographique. En effet, il apparaît clairement que les interactions entre les différents fils d'exécution jouent un rôle majeur. Ces interactions semblent intuitivement être de nature complètement différente lorsque les fils d'exécutions sont distribués sur plusieurs machines et lorsqu'ils s'exécutent sur la même machine.

Définition 2.3 : ***système distribué*** **—** Nous qualifions un système de *distribué* s'il met en jeu plusieurs ressources de calcul qui n'ont pas de mémoire physiquement partagée.

Nous remarquons qu'un système concurrent peut ne pas être distribué, et qu'un système distribué peut ne pas être concurrent (si l'activité n'est pas simultanée).

Nous définissons maintenant des notions utilisées pour décrire la façon de programmer et d'utiliser des systèmes concurrents et/ou distribués.

Définition 2.4 : ***paradigme*** **—** Au sens large, un *paradigme* est un modèle auquel on apparente une famille de modèles de programmation, de modèles de communication, ou d'autres types de modèles ; la distinction se fait selon certains critères fixés.

Il est courant de classer les exécutifs selon le paradigme de *programmation* et le paradigme de *communication*. Dans la suite de ce document, quand nous ne précisons pas de quel paradigme il s'agit, nous parlons par défaut du paradigme de programmation.

Définition 2.5 : ***paradigme de programmation (modèle de programmation)*** **—** Le paradigme de programmation est le point de vue du programmeur sur un système concurrent. Les deux principaux paradigmes de programmation sont *réparti* et *parallèle*. Le paradigme parallèle peut être subdivisé en *parallèle à mémoire distribuée* et *parallèle à mémoire partagée*. Ces paradigmes sont définis et décrits dans les sections suivantes.

Définition 2.6 : ***paradigme de communication*** **—** Le paradigme de communication modélise la façon dont les différents fils d'exécution échangent des informations. Les paradigmes de communication étudiés dans ce document sont le *passage de messages*, l'*invocation de fonction ou méthode à distance*, et la *mémoire partagée*.

2.1.2 Problématiques des systèmes concurrents

Dans cette section, nous présentons les problématiques amenées par les systèmes concurrents. Ces problématiques nous serviront de critères de base à notre étude des systèmes parallèles et répartis. Nous introduisons successivement les notion suivantes : classification des systèmes concurrents, répartition géographique, problématiques des communications, sécurité, performance, et hétérogénéité.

Classes de concurrence : taxonomie de Flynn. La classification qui sert de référence pour les systèmes concurrents est la taxonomie établie par Michael Flynn [75] en 1972, qui distingue les systèmes en fonction des flots d'instructions et des flots de données. Les systèmes sont classés en quatre catégories :

SISD (*Single Instruction, Single Data*) — C'est le modèle traditionnel des machines séquentielles, connu sous le nom de "machine de von Neumann" [179], avec programme et données stockés en mémoire.

SIMD (*Single Instruction, Multiple Data*) — Les différents fils d'exécutions exécutent le même flot d'instructions sur des jeux de données différents. C'est le modèle des *MasPar* et *Thinking Machine* — plusieurs processeurs exécutent les mêmes instructions en même temps sur des données différentes —, mais aussi celui mis en œuvre à l'intérieur des processeurs vectoriels tels que le *Cray*-1 ou plus récemment les unités MMX TM— plusieurs unités à l'intérieur d'un processeur exécutent la même instruction sur plusieurs données.

MISD (*Multiple Instruction, Single Data*) — Plusieurs instructions travaillent sur le même flux de données ; c'est le modèle connu également sous le nom de *pipeline*.

MIMD (*Multiple Instruction, Multiple Data*) — Des flots d'instructions différents travaillent sur des données différentes ; c'est le modèle de la plupart des machines parallèles, supercalculateurs ou grappes, et de la plupart des système répartis.

Répartition géographique. La répartition géographique est un aspect fondamental des systèmes concurrents. Le principal problème de la répartition géographique réside dans l'accès aux données. Le problème peut se situer à plusieurs niveaux :

- stockage : des données sont stockées à un endroit (sur un disque par exemple), et manipulées à un autre endroit. La problématique est alors le *transfert* des données.
- mémoire : lors de l'exécution, deux fils d'exécution partagent et modifient des données communes. Il s'ajoute alors la problématique de la *cohérence* des données lors de manipulations concurrentes.

Le degré de répartition géographique peut être décrit par différentes classes usuelles :

- machines multi-processeur (dites "SMP" — *Symmetric Multi-Processor*) : le stockage et la mémoire sont partagés ;
- machines parallèles à mémoire distribuée, grappes avec NFS : la mémoire n'est pas partagée, le stockage des données est partagé ;
- machines distinctes : ni la mémoire ni le stockage ne sont partagés ;

Un degré supplémentaire est atteint pour des machines faisant partie de *sites* différents, c'est-à-dire des domaines d'administration ayant des administrateurs différents pour les ressources informatiques, des utilisateurs différents, des politiques différentes.

Interconnexions, communications. Les différentes tâches d'un système concurrent interagissent pour parvenir à leur but commun. D'un point de vue général, nous nommons ces interactions — tout échange d'information entre deux tâches — une *communication*. La communication peut être matérialisée sous la forme d'un échange de messages sur un réseau entre deux machines, une communication "locale" de type *tube* ou *socket* Unix entre deux processus sur la même machine, ou encore un échange par segment de mémoire partagée de type IPC System V entre des processus ou *threads* sur une même

machine. Les réseaux sont couramment classés en catégories (dites *Network Area*) selon la distance couverte. Les principales classes utilisées en calcul scientifique sont :

WAN (*Wide Area Network*, réseau longue distance) — C'est une classe de réseau qui couvre de longues distances, allant de quelques kilomètres à l'échelle mondiale. Le but d'un WAN est généralement de relier des réseaux plus petits. Le WAN le plus connu est probablement *internet* mais il en existe d'autres. Une des caractéristiques principales des WAN est qu'ils sont gérés par des organisations différentes. Des exemples de technologies utilisées dans les WAN sont ATM [80], SDH/SONET [176], FDDI [76] ou *Frame Relay* [77] ; de plus en plus souvent, le protocole utilisé au-dessus est IP [150].

LAN (*Local Area Network*, réseau local) — C'est une classe de réseau dont la portée est locale, c'est-à-dire un bureau, un immeuble, ou une maison. La plupart du temps, de tels réseaux sont gérés par une seule personne ou une seule organisation. Les technologies utilisées sont par exemple Ethernet [142] ou *Token Ring* [143], avec par exemple les protocoles IP ou IPX.

SAN (*System Area Network*, réseau haute performance, ou réseau haut débit) — C'est une classe de réseau dédiée à la construction de *grappes* (*clusters*), dites également "fermes de PC". Leur portée est très restreinte — de l'ordre de quelques mètres la plupart du temps — pour permettre de très hautes performances : haut débit et faible latence. Les technologies utilisées sont par exemple Myrinet [41, 52], SCI [108], InfiniBand [44] ou VIA [70], chacun utilisé à l'aide d'un protocole spécifique, rarement IP.

La connectivité proposée par les réseaux est variable. La connectivité dépend de :
- stratégies de routage : le choix de la route pour relier deux points du réseau.
- pare-feu : pour éviter les intrusions dans les systèmes, des pare-feux (*firewall*) sont mis en place. Un pare-feu empêche d'établir des connexions vers un ensemble de machines.
- filtrage : certains pare-feux "intelligents" choisissent, en fonction des services autorisés, les connexions à laisser passer.

Sécurité. La sécurité sur les réseaux est une autre problématique importante. Dans certains cas — applications militaires, secrets industriels, applications critiques — elle est indispensable. Dans des cas moins critiques, elle n'est pas requise. La sécurité peut être décomposée en trois aspects :

authentification — l'authentification consiste en une reconnaissance mutuelle de l'identité des parties en présence. Il s'agit de savoir "à qui l'on parle".

confidentialité — la confidentialité consiste en la non-divulgation d'informations à une tierce partie. Typiquement, la confidentialité peut être rompue sur un WAN si une tierce partie intercepte une communication. La confidentialité des données est assurée par des techniques cryptographiques basées sur le *chiffrement* (appelées parfois *cryptage* par anglicisme).

intrusion — l'intrusion est la capacité d'une tierce partie à *exécuter* des programmes sur une machine pour laquelle elle n'est pas accréditée.

Performance. La performance est la vitesse d'exécution d'un système. Elle est caractérisée par des grandeurs appelées *métriques*. Pour une application donnée, la métrique usuelle est le temps d'exécution en fonction de la taille du problème — la taille des données données en entrée de l'application. Pour les réseaux, les métriques usuelles sont le débit et la latence. Le débit représente la quantité d'information par unité de temps ; la latence est le temps nécessaire pour transmettre la quantité d'information minimale permise par le réseau — habituellement un octet, un mot machine, ou un message vide quand c'est possible.

Hétérogénéité. L'hétérogénéité consiste en une disparité de type ou de vendeur au niveau du matériel, du système d'exploitation, ou de l'intergiciel. L'hétérogénéité constitue un problème principalement quand les parties d'une part et d'autre d'un lien réseau sont différentes (logiciel et/ou matériel), et utilisent un codage des données différent. L'encodage des données est caractérisé par :
- la taille des mots machine, et donc le nombre de bits qui représentent certains types de données comme les entiers ;

– l'ordre des octets pour les données sur plusieurs octets (entiers et flottants). L'ordre des octets et la taille des mots utilisés par chaque machine dépend du type de son processeur, et parfois de paramètres de configuration. On parle de *little endian* ("petit boutiste") quand la machine stocke l'octet de poids faible en premier, *big endian* ("gros boutiste") dans le cas contraire.
– la correspondance entre les octets et le caractère représenté. La correspondance est établie par une table de caractère telle que ISO-8859, *Unicode*, ASCII ou EBCDIC.

Pour se comprendre, les deux parties doivent être d'accord sur la façon d'encoder les données, de présenter les messages, et la sémantique des messages . Dans ce cas, nous parlons d'*interopérabilité*. L'hétérogénéité peut également se faire sentir au niveau du protocole utilisé pour gérer les connexions. Enfin, l'hétérogénéité intervient au niveau des applications : la plupart du temps, un programme compilé pour être exécuté sur une machine ne pourra pas être exécuté sur une machine de type différent.

Ces différents aspects — classes de concurrence, répartition géographique, communications, sécurité, performance et hétérogénéité — nous serviront comme critères pour étudier les exécutifs que nous introduisons à la section suivante.

2.1.3 Infrastructures logicielles

Nous nous intéressons dans cette section plus particulièrement aux infrastructures logicielles utilisées pour programmer les systèmes concurrents. Nous étudions les approches utilisées dans ces infrastructures logicielles. Nous commençons par des définitions générales pour appréhender la notion d'*exécutif communicant*. Nous nous intéressons ensuite à la notion d'abstraction des ressources ; l'abstraction est une technique utilisée par les exécutifs pour atteindre portabilité et généricité. Nous nous intéressons ensuite à la construction d'exécutifs par composants logiciels, ce qui est une approche légèrement différente de la précédente pour construire des infrastructures logicielles pour systèmes concurrents.

2.1.3.1 Définitions générales

Il existe différents types de logiciels utilisés pour aider la programmation des systèmes concurrents. La plupart du temps, ces logiciels s'insèrent entre le code applicatif et le matériel ou le système d'exploitation. Il est possible de distinguer différentes variétés parmi ces couches logicielles. Il est courant de faire la distinction entre un *intergiciel* (en anglais : *middleware*) dont le but est généralement d'assurer une relative indépendance par rapport au système et de prendre en charge l'aspect distribué, et un *support exécutif* (ou *noyau exécutif*, ou en anglais : *runtime*) dont le but est de mettre en œuvre un modèle d'exécution, par exemple cible d'un compilateur de langage parallèle. Il existe une relative confusion entre les deux — par exemple, il arrive que CORBA soit considéré comme support exécutif ou intergiciel selon que l'on s'intéresse à son modèle objet ou à sa gestion de l'aspect distribué —, et la séparation peut parfois sembler arbitraire. Dans cette section, nous posons les définitions que nous utiliserons tout au long de ce document.

Définition 2.7 : ***intergiciel (logiciel médiateur,* middleware*)*** — Un intergiciel est une infrastructure logicielle qui sert d'*intermédiaire* entre une application et le système. Les buts sont variés, par exemple assurer la portabilité sur plusieurs systèmes d'exploitation ou pilotes réseaux, ou cacher l'aspect distribué. Un intergiciel réalise un modèle de programmation de l'architecture dans laquelle il s'inscrit.

Définition 2.8 : ***exécutif (support exécutif,* runtime*)*** — Un exécutif est la partie d'un intergiciel constituée du code chargé et utilisé par les applications pour mettre en œuvre le modèle lors de l'exécution.

L'intergiciel est l'infrastructure globale, l'exécutif est la partie chargée à l'exécution. Un exemple d'intergiciel assurant la gestion de l'aspect distribué est représenté à la figure 2.1. Par exemple, CORBA [146] de l'OMG, HLA [109] du DMSO, le JDK [97] de *Sun Microsystems* sont des intergiciels ; leurs exécutifs correspondants sont un ORB, un RTI HLA, une JVM. Une *application* (ou *code applicatif* ou *code métier*) est la partie d'un programme qui réalise directement les actions qui sont la finalité du programme, à l'inverse d'un intergiciel dont le but est d'aider l'application à utiliser les ressources.

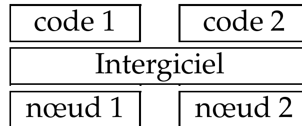

FIG. 2.1 – *Un intergiciel pour cacher la répartition géographique des ressources.*

Exécutifs communicants. Cette étude est centrée essentiellement sur les exécutifs communicants. Nous utiliserons certains termes avec un sens particulier ou plus général que leur sens habituel ; nous les définissons ici. Étant donné que la distinction entre *exécutif* et *intergiciel* établie par les définitions 2.8 et 2.7 n'est pas toujours très nette et importe peu dans notre analyse, nous désignerons toute couche logicielle s'insérant entre le code applicatif et le système, et étant utilisée *directement par l'application*, par le terme générique "*exécutif*".

Définition 2.9 : ***environnement, plate-forme de communications*** — Comme nous le verrons dans la section sur l'abstraction, il existe des couches logicielles utilisées par les *exécutifs* (au sens de la définition 2.8) pour accéder au système. Ces couches logicielles qui ne sont pas destinées à être utilisées directement par le code applicatif seront désignées par les termes *environnement* ou *plate-forme de communication* selon leur finalité.

Définition 2.10 : ***exécutif communicant*** — Nous nous restreignons aux exécutifs qui réalisent des communications, que ce soit pour transmettre explicitement des données pour le compte d'une application (exemple : `MPI_Send`) ou que les communications soient implicites (exemple : `MPI_Barrier`). Tout exécutif utilisant des mécanismes de communication sera désigné par le terme "*exécutif communicant*".

Chacune des deux sections suivantes décrit une approche de conception des exécutifs communicants : d'abord l'abstraction des ressources, puis les approches par composants. Dans la suite de ce document, par convention nous utilisons sur les figures une police de caractère différente pour distinguer les implémentations et les *interfaces*.

2.1.3.2 Abstraction des ressources

Les exécutifs communicants utilisent un réseau ou tout autre moyen de communication. Il existe de nombreux types de réseaux qui diffèrent selon le constructeur, le modèle, la version, etc. Or, chaque type de réseau particulier propose une interface de programmation qui lui est propre — XTI [104] ou *sockets* [38] BSD pour les réseaux IP [150], SISCI [95] pour SCI [108], BIP [156], *Myrinet Express* (MX) [137], ou GM [136] pour Myrinet [41, 52], IB Access [107] pour InfiniBand [44]. Ces interfaces de programmation sont fournies par le système d'exploitation, par un pilote spécifique, ou encore par une bibliothèque, avec des approches parfois radicalement différentes les unes des autres. Ceci introduit donc un lien très fort entre une implémentation particulière d'un exécutif et un un type particulier de réseau.

Cette approche basique qui consiste à développer n versions d'un exécutif pour n réseaux différents est illustrée par la figure 2.2. C'est l'approche retenue par exemple pour les plates-formes *Fast Messages* [148] et *Active Messages* [178] : les versions dédiées à Myrinet, TCP ou UDP sont entièrement distinctes. Cette approche résulte en une duplication de code et d'efforts : la version de l'exécutif A sur le réseau 1 a probablement beaucoup en commun avec le même exécutif sur le réseau 2.

L'*abstraction* est une technique mise en œuvre par de nombreux exécutifs pour pallier à ce problème. L'abstraction consiste en la définition d'une certaine interface pouvant correspondre à plusieurs incarnations. C'est une façon de rendre l'interface de programmation indépendante du type de réseau.

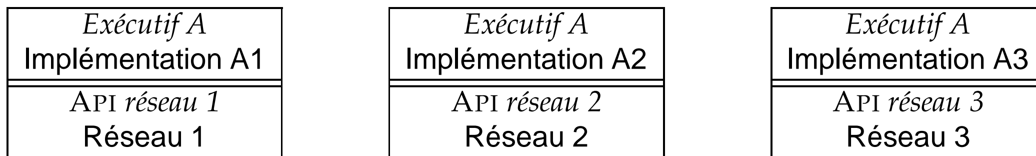

FIG. 2.2 – *Implémentations d'exécutifs liées au réseau : une implémentation différente pour chaque type de réseau.*

Portabilité par pilotes. Pour éviter la duplication de code qui résulte de l'écriture de n versions d'exécutifs pour n réseaux, la solution couramment retenue est basée sur un découplage entre le bas niveau et le haut niveau. Dans ce contexte, nous entendons par bas niveau l'interface de programmation d'un type de réseau particulier, et par haut niveau l'exécutif lui-même. Il s'agit de distinguer une partie générale réutilisable — haut niveau — qui implémente les fonctionnalités de l'exécutif fournies aux applications, et une partie dépendant de matériels spécifiques — bas niveau. La partie haute factorise le code que des versions spécifiques ont en commun. Des modules logiciels appelé *pilotes* (en anglais : *driver*) réalisent l'adaptation à chaque réseau. Ces pilotes réalisent une *abstraction*, c'est-à-dire qu'ils présentent une interface de programmation constante, quel que soit le type de réseau réellement utilisé au niveau bas. La partie haute de l'exécutif utilise toujours cette interface abstraite. Un exécutif devient alors *portable* sur un certain nombre de réseaux différents, avec la portabilité définie comme suit.

Définition 2.11 : *portabilité* — Nous appelons *portabilité* la capacité d'un code à utiliser différentes variantes d'un type donné de ressource. Par exemple, un exécutif est *portable* par rapport au réseau s'il est capable d'utiliser différents types de réseaux.

Sur l'exemple de la figure 2.3, l'exécutif A est implémenté par A/PA qui utilise toujours l'interface abstraite *PA*. La partie dépendant du type de réseau est isolée dans les pilotes PA1, PA2 et PA3 qui présentent tous la même interface *PA*. Un exemple où la portabilité est assurée par des pilotes distincts d'une partie générale est l'interface abstraite de périphérique (*Abstract Device Interface*, ou ADI [101]) de MPICH [100, 102].

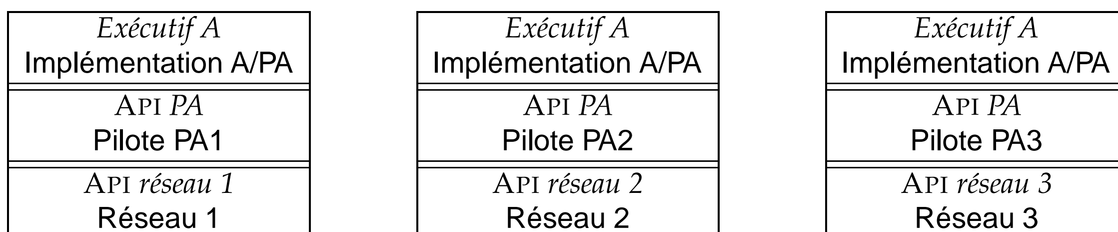

FIG. 2.3 – *Portabilité assurée par des pilotes : la même implémentation* A/PA *est réutilisée d'un réseau à l'autre.*

Généricité. La portabilité par pilotes à l'intérieur des exécutifs constitue une première étape d'abstraction. Elle évite la duplication d'une grande partie de code : pour porter un exécutif sur n réseaux, il suffit d'écrire une partie générique et n pilotes, chaque pilote représentant une quantité de travail bien moindre que le développement d'un exécutif complet.

Cependant, étant donnée la variété d'exécutifs existants, il est judicieux de considérer le problème de porter m exécutifs sur n réseaux. Pour une portabilité par pilotes, il est nécessaire de développer $m \times n$ pilotes différents. Même s'il ne s'agit plus de développer $m \times n$ versions d'exécutifs complets comme il serait nécessaire avec l'approche basique, la multiplication du nombre de pilotes incite cependant à chercher plus de généricité. L'idée consiste à réutiliser pour plusieurs exécutifs un pilote donné, de façon à n'écrire qu'une seule fois un pilote pour un réseau donné. Dans l'exemple de la

figure 2.3, les pilotes offrent l'interface *PA* qui est spécifique pour l'exécutif *A*. Cependant, il est probable que des pilotes pour PA1 et PB1 pour deux exécutifs *A* et *b* aient une philosophie très proche et une grande partie de code similaire ; il serait souhaitable de de mettre en commun le code qui réalise les opérations semblables.

Une solution consiste en la généralisation du découplage entre le haut et le bas niveau. Cette généralisation se traduit par l'insertion d'une couche logicielle supplémentaire entre les deux niveaux. Cette couche logicielle, communément appelée *environnement de portabilité*, permet de séparer une implémentation d'un pilote d'un exécutif donné. Elle fournit une interface apte à être utilisée par plusieurs exécutifs. Comme l'illustre la figure 2.4, les exécutifs utilisent l'interface *générique G* fournie par l'environnement de communications ; plusieurs exécutifs, *A* et *B* utilisent la même interface, et donc les même pilotes. Ces exécutifs basés sur cette interface générique sont alors portables sur tous les types de réseaux pris en charge par l'environnement.

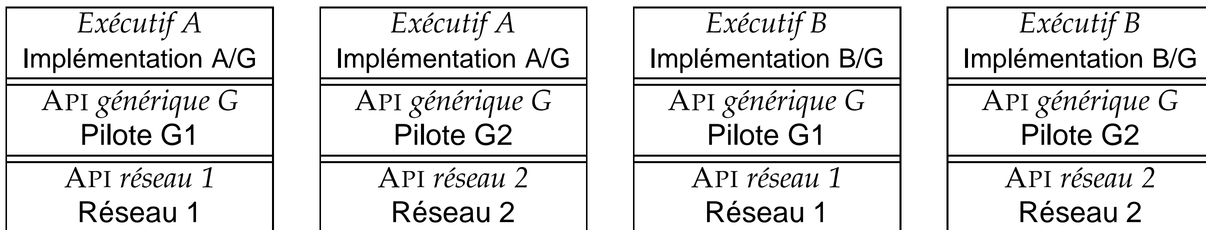

FIG. 2.4 – *Implémentations d'exécutifs sur un environnement générique de communication : plusieurs exécutifs réutilisent les mêmes infrastructures de portabilité.*

Définition 2.12 : *générique, spécifique* — La *généricité* est la capacité d'un environnement de communication portable à pouvoir être utilisé par plusieurs exécutifs différents. Dans le cas contraire, l'environnement de communication est qualifié de *spécifique*.

Des exemples d'environnements génériques sont Madeleine [8] ou ADAPTIVE [164].

Notons que la plupart de ces environnements combinent des notions de *portabilité* (utiliser plusieurs types de réseaux) et de *généricité* (être utilisable par divers exécutifs). Ces environnements sont souvent eux-mêmes conçus sur le modèle des exécutifs portables : une partie générique regroupe les fonctions communes quel que soit le réseau, et des pilotes spécifiques prennent en charge l'adaptation aux réseaux. Pour disposer de m exécutifs sur n réseaux différents, il suffit dans cette approche d'écrire les m exécutifs s'appuyant sur une unique couche commune, la couche commune, et les n pilotes. Il s'agit donc de développer $m + n$ codes au lieu de $m \times n$.

2.1.3.3 Approches par composants

L'approche par composants est une autre façon de concevoir plusieurs exécutifs portables en évitant de dupliquer des efforts et en réutilisant du code qui a fait ses preuves. Cependant, au lieu de considérer que les divers exécutifs sont des variantes d'un mécanisme général comme suggéré par l'approche l'abstraction des ressources, l'approche par composants considère que ce sont des variantes d'assemblage des même briques de base.

Il existe une diversité d'exécutifs et il est vraisemblable qu'il y aura encore à l'avenir, avec la popularisation des grilles et du calcul distribué, de nombreux exécutifs chacun adapté à une utilisation particulière. Pourtant, la plupart des exécutifs communicants mettent en jeu des mécanismes communs : nommer les ressources, échanger des données, détecter des erreurs. Dans la pratique, il existe des différences dans le domaine d'utilisation, le langage de programmation ou le modèle de programmation, mais fondamentalement les techniques de base sont les mêmes. Par exemple MPI [79] est un mécanisme de communication par messages avec un adressage par numérotation des processus, et met l'accent sur la performance ; CORBA [146] fonctionne par appel de méthode, l'adressage se fait par la notion d'objets, et l'accent est mis plutôt sur l'interopérabilité et la robustesse. Mais en essence, ce ne sont que deux instanciations des mécanismes génériques communs aux exécutifs communicants.

Puisque les exécutifs communicants sont semblables mais pas identiques, il semble judicieux de factoriser les parties communes qui sont basées sur les mêmes notions, pour éviter de refaire plusieurs fois le même travail, accélérer le développement et réutiliser du code qui a fait ses preuves. C'est le but de l'approche *générique* décrite ci-dessus : une base commune regroupe le code commun factorisé sur laquelle on ajoute des parties spécifiques pour obtenir les différents exécutifs. Cette approche est pertinente dans les cas où l'on distingue une version générale du problème à résoudre, de laquelle on peut dériver des versions spécialisées proches.

Cependant, beaucoup d'exécutifs ne sont pas des *variantes d'un mécanisme* général, mais plutôt des *variantes d'assemblage* des mêmes éléments, seules les briques de base (nommage des ressources, échange de messages, mécanismes d'établissement de connexion) étant les mêmes, pas l'architecture globale selon laquelle ces briques sont assemblées. Une approche adaptée pour factoriser les notions et le code communs consiste en l'utilisation de *composants logiciels* pour la construction des exécutifs. La notion de composant logiciel a été définie par Clemens Szyperski [173], mais d'autres définitions légèrement différentes existent selon les références. Nous nous baserons sur la définition suivante :

Définition 2.13 : ***composant*** — Un composant est une unité binaire indépendante de déploiement qui interagit avec son environnement par l'intermédiaire d'interfaces bien spécifiées, appelées *ports*. L'assemblage des composants est effectué par une tierce partie.

Dans les cas où il ne s'agit pas de composants logiciels au sens strict du terme (pas composables par une tierce partie, par exemple), nous emploierons plutôt le mot "*modules*".

Il existe des plates-formes [169, 73, 106, 121] dédiées à la construction d'exécutifs par composants logiciels. De telles plates-formes se basent sur le principe de la décomposition d'un problème ou d'un logiciel complexe et/ou de grande taille en entités *composants*. Le logiciel global — l'exécutif — est alors réalisé en interconnectant les composants. L'architecture de tels ensembles est constituée d'une part d'une plate-forme d'exécution qui réalise à proprement parler l'assemblage des composants, d'une bibliothèque de composants prêts à être assemblés, et d'assemblages préprogrammés qui incarnent selon les cas des exécutifs ou des protocoles réseau.

Les approches par abstraction et par composants ne sont pas opposées. Il est possible de réaliser une plate-forme de communication générique à l'aide de composants ; par exemple, chaque pilote d'une telle plate-forme peut être considéré comme un composant.

2.2 Les systèmes parallèles

Dans cette section, nous nous intéressons à un type particulier de systèmes concurrents : les *systèmes parallèles*. Les systèmes parallèles ont pour origine la recherche de la haute performance. Transformer un programme séquentiel en programme parallèle est certes une tâche complexe, mais le prix de la complexité est contrebalancé par la performance gagnée. La définition précise du calcul parallèle est délicate ; il est en effet difficile de trouver une définition qui englobe tout et seulement le calcul parallèle [1]. Nous posons la définition suivante qui sera celle de référence pour la suite de ce document :

Définition 2.14 : ***calcul parallèle*** — Le calcul parallèle consiste en le découpage d'un programme en plusieurs tâches qui peuvent être exécutées en même temps dans le but d'améliorer le temps global d'exécution du programme.

Nous remarquons que cette définition recouvre des cas où la haute performance n'est pas *a priori* le but principal ; par exemple lorsqu'un programme est parallélisé dans le seul but de pouvoir profiter d'une plus grande quantité de mémoire, il s'agit bien d'améliorer la vitesse d'exécution, car sans parallélisation l'exécution sur une seule machine qui ne dispose pas de suffisamment de mémoire est confrontée à la pagination (*swap*) qui dégrade très fortement les performances. Que la parallélisation serve à utiliser un plus grand nombre de processeurs ou une plus grande quantité de mémoire, l'objectif est d'améliorer les performances globales du programme.

1. À part peut-être la définition trouvée sur la FAQ de `comp.parallel` : "*It's like a parallel computer: some parts are going to work and others won't !*"

Il est courant de distinguer deux types de calcul parallèle : le calcul parallèle à mémoire distribuée et le calcul parallèle à mémoire partagée, qui se définissent comme suit :

Définition 2.15 : ***calcul parallèle à mémoire distribué*** — Le calcul parallèle à mémoire distribuée est le sous-ensemble du calcul parallèle (selon la définition 2.14) qui satisfait aussi la définition 2.3 (système distribué).

Définition 2.16 : ***calcul parallèle à mémoire partagée*** — Le calcul parallèle à mémoire partagée est le sous-ensemble du calcul parallèle (selon la définition 2.14) qui ne satisfait pas la définition 2.3 (système distribué).

Dans la suite de cette section, nous nous intéressons d'abord aux architectures parallèles, puis nous décrivons les modèles de programmation du parallélisme, et enfin nous effectuons un tour d'horizon des infrastructures logicielles utilisées en parallélisme.

2.2.1 Architectures parallèles

Des architectures spécifiques de machines et de réseaux ont été développées pour le calcul parallèle. La plupart du temps, ces machines sont dédiées au calcul parallèle soit par conception, soit par configuration.

Les architectures parallèles peuvent être caractérisées selon les critères donnés en section 2.1.2 : habituellement, une machine parallèle est une plate-forme homogène, dont tous les nœuds présentent la même architecture, géographiquement proches, à l'intérieur d'un seul domaine d'administration. L'ensemble des nœuds est vu comme une seule entité logique, tous les nœuds sont équivalents. Par conséquent, les infrastructures de sécurité entre les différents nœuds sont pratiquement inexistantes ; on suppose en général qu'une machine parallèle est à l'abri derrière un pare-feu qui l'isole des intrusions et interceptions. Les réseaux couvrent de petites distances, sont rapides, dédiés au parallélisme. Le maître mot est la haute performance.

Machines parallèles. Historiquement, les premières machines parallèles à être diffusées commercialement ont été des architectures appelées communément *supercalculateurs*. Les supercalculateurs sont des machines caractérisées par un grand nombre de processeurs reliés par un réseau interne à haute performance — machines dites "massivement parallèles" —, ou des processeurs exploitant le parallélisme en interne capables de travailler directement sur des vecteurs de nombres — on les qualifie alors de "vectorielles" —, un système d'exploitation souvent spécifique, et un prix invariablement exorbitant ! On associe souvent les supercalculateurs à Seymour Cray et des machines telles que le *Cray*-1 ou le *Cray* X-MP. D'autres constructeurs comme par exemple *Nec*, *Fujitsu*, *nCube*, *Convex* ou *Alliant* fabriquaient également des supercalculateurs et des mini-calculateurs, versions réduites de supercalculateurs à un tarif plus abordable.

L'ère des supercalculateurs a connu son apogée depuis les années 1970 jusqu'au début des années 1990. Par la suite, l'augmentation de la puissance de calcul des processeurs dits "grand public" et l'avènement des grappes ont relégué beaucoup des supercalculateurs au rang de pièces de musées. Certains constructeurs de supercalculateurs se sont alors tournés vers les grappes, d'autres continuent sur le marché restreint, beaucoup ont fini en faillite. Certaines des idées développées à l'origine pour les supercalculateurs sont cependant passées dans les machines "grand public", par exemple des unités vectorielles ont été intégrées dans pratiquement tous les processeurs pour ordinateurs personnels contemporains (ce sont les MMX™ et SSE™ chez *Intel*, *3D Now!*™ chez *Advanced Micro Devices*, ou *Altivec*™ chez *Motorola*), et les serveurs d'entrée de gamme utilisent couramment les techniques SMP avec un nombre restreint de processeurs.

Grappes. L'émergence des grappes de calcul est liée à la puissance sans cesse croissante des PC standard, comblant le retard qui les séparait des processeurs des supercalculateurs. Des constructeurs comme *Myricom* ou *Dolphin* ont alors proposé vers le milieu des années 1990 des matériels réseaux de classe SAN à très haute performance tels que Myrinet [41, 52] et SCI [108], de façon à rattraper les

supercalculateurs également dans le domaine des interconnexions. Ces réseaux offrent de très hautes performances, du même ordre de grandeur que celles disponibles directement sur le bus système des PC. De simples PC interconnectés par ces réseaux, même s'ils ne sont pas capables de rivaliser en performances pures avec les supercalculateurs, les battent indubitablement en ce qui concerne le rapport performance/prix !

Depuis, de nombreux fabriquants de grappes "prêtes à l'emploi" sont apparus et proposent des configurations complètes constituées majoritairement de composants standard, donc à un tarif abordable, généralement équipées du système d'exploitation Linux. Les grappes détrônent peu à peu les supercalculateurs. Tendance significative, environ 30 % des 500 machines les plus puissantes du monde sont maintenant des grappes (149 sur 500) dans la 21ème édition du Top 500 [34] de juin 2003, alors qu'elles n'y ont fait leur première apparition qu'en juin 1997.

2.2.2 Modèles de programmation parallèle

Dans cette section, nous décrivons les modèles de programmation utilisés pour programmer les architectures parallèles que nous venons de présenter. La façon de programmer de tels systèmes est souvent intimement liée à leur architecture. Nous nous focalisons sur la programmation des systèmes MIMD. La plupart du temps, de tels systèmes parallèles basés sur des flots d'instructions différents exécutent en réalité le même programme. Ce modèle d'exécution, cas particulier de MIMD, a été baptisé "SPMD" par Frederica Darema [66]. Il existe deux modèles de programmation courants en parallélisme, souvent de type SPMD : à mémoire distribuée (définition 2.15) ou à mémoire partagée (définition 2.16).

Parallélisme à mémoire distribuée. En parallélisme à mémoire distribuée, la machine parallèle est vue comme un ensemble de nœuds équivalents les uns aux autres ; chaque nœud est constitué d'un (ou plusieurs) processeur et de sa mémoire associée. Les mémoires des différents nœuds sont séparées ; pour accéder aux données des autres nœuds, il est nécessaire de réaliser des communications. Les deux paradigmes de communication principaux sont le passage de messages et l'appel de procédure à distance (RPC — *Remote Procedure Call*).

En passage de messages, les communications peuvent avoir lieu en *point à point* — un émetteur, un récepteur — ou peuvent être *collectives*, c'est-à-dire impliquer plusieurs nœuds. Lors d'un passage de message en point à point, l'émetteur envoie les données, le récepteur les reçoit explicitement. Les opérations collectives sont de trois types : synchronisation, communications, et calculs. La synchronisation est essentiellement une barrière, où tous les nœuds attendent tous les autres. Les communications collectives sont : la diffusion (*broadcast*) — la même donnée envoyée par un nœud à tous les autres —, dispersion/regroupement (*scatter/gather*) — dispersion et regroupement de blocs d'un vecteur entre un nœud et tous les autres —, ou d'autres schémas plus évolués tels que des opérations "tous vers tous" (*all-to-all*). Les calculs collectifs sont typiquement la réduction (*reduce*) qui permet par exemple de récupérer sur un nœud la somme d'un vecteur distribué sur tous les nœuds.

En RPC, l'opération de base est l'invocation d'une procédure à distance, c'est-à-dire qu'un nœud peut appeler une procédure sur un autre nœud. L'invocation peut contenir des paramètres et la procédure peut également fournir des valeurs de retour. L'intérêt du modèle de programmation par RPC est qu'en plus de transférer des données, il transfère également le flot de contrôle.

Il existe un modèle à mi-chemin entre passage de messages et RPC appelé "messages actifs" (*active messages*). En messages actifs, l'envoi se fait de la même façon qu'en passage de messages. La réception quant à elle se fait par un traitant (*handler*) appelé quand une arrivée de message est détectée ; la réception se fait ensuite à l'intérieur du traitant, en utilisant une méthode similaire à celle employée en passage de messages. Les messages actifs sont cependant rarement un modèle de programmation applicatif, mais plutôt un modèle mis en œuvre par des plates-formes de communication destinées à être utilisées par des exécutifs.

Parallélisme à mémoire partagée. En parallélisme à mémoire partagée, la machine parallèle est vue comme un ensemble de processeurs qui accèdent à la même mémoire centrale. La mémoire peut être

physiquement partagée par plusieurs processeurs (SMP), ou une mémoire physiquement distribuée avec l'illusion d'une mémoire partagée assurée par des mécanismes matériels (NUMA) ou logiciels (DSM).

En mémoire partagée, l'entité de base est un fil d'exécution, matérialisé par un *thread* à l'exécution. Les échanges de données entre les différents fils se font simplement par des lectures/écritures vers la même zone de la mémoire. Ceci amène le principal aspect de la programmation parallèle à mémoire partagée : la cohérence. Il s'agit de spécifier dans quelle mesure les différents fils d'exécution "voient" les modifications apportées par les autres sur une zone mémoire partagée. En effet, il est rare que tous les *threads* voient réellement directement la même mémoire ; dans le cas de DSM ou de NUMA, le partage est ajouté au-dessus d'une mémoire physiquement distribuée, et même en SMP les processeurs ont des caches qui peuvent mener à l'existence à un moment donné de plusieurs versions d'une donnée. Les garanties sur la propagation des actions sur la mémoire sont spécifiées par un *modèle de cohérence* [127]. La façon de gérer la propagation des modifications de façon à respecter un modèle de cohérence s'appelle un *protocole de cohérence*. La parallélisation des applications n'étant pas toujours chose aisée, des langages et compilateurs spécialisés ont vu le jour pour faciliter la programmation parallèle à mémoire partagée. Le parallélisme est géré directement dans le langage ; le compilateur génère un code parallèle en suivant les indications données par le programmeur.

2.2.3 Infrastructures logicielles pour le parallélisme

Nous présentons maintenant les infrastructures logicielles qui implémentent les modèles de programmations que nous venons d'introduire. Dans cette section, nous réalisons un tour d'horizon des infrastructures logicielles utilisées pour programmer les architectures parallèles. Ces infrastructures sont de quatre types : les plates-formes génériques de communication utilisées par les exécutifs, les exécutifs à passage de messages, les mémoires virtuellement partagées, et les compilateurs pour langages parallèles.

2.2.3.1 Plates-formes génériques de communication pour le parallélisme

Nous présentons les plates-formes génériques de communication pour le parallélisme. De telles plates-formes sont destinées à être utilisées par des exécutifs et servent de fondations aux exécutifs décrits dans les sections suivantes. Elles leur apportent des infrastructures de portabilité par rapport aux réseaux, et sont réutilisables par plusieurs exécutifs.

Ces plates-formes ont en commun une conception guidée par les principes fondamentaux du parallélisme, à savoir la haute performance avant tout. Ainsi, les copies en mémoire sont évitées autant que possible pour favoriser le débit. La topologie est gérée de façon statique — construite une fois pour toutes à l'initialisation. Ce choix est pertinent dans le cas d'une grappe, car la configuration change rarement en cours de session. La simplicité de mise en place est alors un atout. La configuration gérée est homogène du point de vue de l'architecture du processeur et du réseau.

Berkeley Active Messages. *Active Messages* [133, 178, 131] est une plate-forme de communication développée sous la direction de David Culler au milieu des années 1990 à l'université de Berkeley. *Active Messages* a été conçu dès le départ comme une plate-forme de communication minimaliste pour offrir la plus haute performance possible : les messages sont calibrés et limités en taille, et la réception se fait par messages actifs, c'est-à-dire que quand un message arrive, un traitant est appelé pour gérer la réception ; l'initiative de réception d'un message n'est pas prise par l'application mais est déclenchée implicitement par l'arrivée d'un message. Beaucoup de contraintes sont imposées dans le but d'exploiter au maximum le matériel pour d'excellentes performances brutes, parfois au détriment de la facilité d'utilisation. *Active Messages* a été porté sur ATM, Myrinet, TCP, avec une implémentation de référence sur UDP. *Active Messages* a été utilisé notamment pour offrir une interface *sockets* [160], une implémentation de MPI, et le système de fichiers distribué xFS. *Active Messages* est aujourd'hui dépassé par d'autres plates-formes qui n'ont pas les limitations qu'avait *Active Messages*, mais l'idée des messages actifs a été reprise dans de nombreuses plates-formes.

Illinois *Fast Messages*. *Fast Messages* 2.0 [148] est une plate-forme de communication bas-niveau, développée sous la direction d'Andrew Chien dans le groupe de recherche CSAG de l'université de l'Illinois à Urbana-Champaign. *Fast Messages* est largement inspiré des principes d'*Active Messages*, basé sur des messages actifs. Une attention toute particulière a été donnée aux petits messages, suite à l'observation des tailles typiques de messages en parallélisme. *Fast Messages* propose une interface de très bas niveau, restreinte à seulement cinq fonctions. *Fast Messages* a été porté sur TCP, Myrinet et *Cray* T3D, et a été utilisé notamment pour l'implémentation MPI MPI-FM et une version de PVM.

Madeleine 2 et 3. Madeleine [47, 8] est une bibliothèque de communications dédiée aux réseaux hautes performances présents sur les grappes, développée par Olivier Aumage au LIP (Lyon). Elle fait partie de l'environnement de programmation parallèle PM^2 [139, 29] de Raymond Namyst ; PM^2 inclut également la bibliothèque de *multi-threading* Marcel [138]. Dans sa deuxième version, Madeleine est capable d'exploiter simultanément plusieurs protocoles parmi SISCI [95] (pour SCI [108]), BIP [156] (pour Myrinet [52]), TCP [151], VIA [70], SBP [162] (pour un accès direct à Ethernet [142]) et MPI [79]. La troisième version [48] de Madeleine ajoute le support d'interconnexions de grappes ("grappes de grappes").

Dans Madeleine, les communications sont organisées autour de canaux de communication. Un *canal* au sens Madeleine est une structure de communication dans laquelle tous les nœuds sont reliés à tous les autres, à la façon d'un *communicateur* MPI. Par conception, Madeleine 2 impose que tous les nœuds soient reliés par le même réseau. Madeleine 3 permet une certaine hétérogénéité au niveau des réseaux, et assure le passage des messages d'un réseau à un autre.

L'échange de messages avec Madeleine est réalisé à l'aide de primitives du type *pack*/*unpack*. Le message est construit de manière incrémentale au fur et à mesure des opérations *pack*. Madeleine choisit le moment et la méthode pour transmettre les données en fonction d'indications fournies par l'utilisateur. À la différence de nombreux autres environnement, les indications sont données au niveau sémantique et non au niveau fonctionnel. Cette interface a été conçue dans le but de permettre des transferts zéro-copie même pour des messages de longueur variable non-connue à l'avance. Les performances obtenues sont excellentes ; Madeleine exploite plus de 95 % du débit nominal du matériel sur Myrinet, Myrinet-2000 et SCI. Madeleine a notamment été utilisée par l'implémentation MPI MPICH/*Madeleine* [46].

Panda et Ibis. Panda [161] est une plate-forme de portabilité pour des exécutifs parallèles, développée à l'université d'Amsterdam dans l'équipe de recherche d'Henri Bal. Panda prend en charge la portabilité par rapport aux réseaux (principalement Myrinet et TCP) et au *multi-threading*. Des versions de PVM, MPI, et *Orca* (langage à parallélisme de données) ont été portées sur Panda. Panda est sur le point de devenir obsolète.

La suite des travaux sur Panda est Ibis [177], un environnement de communications en Java. L'utilisation du langage Java apporte la portabilité, la facilité de dynamicité, et le principe de RPC (RMI, en Java) dans le langage. Ibis implémente des RMI rapides sur TCP et Myrinet, la possibilité de réaliser des d'invocations vers des groupes, et un modèle de programmation qui combine MPI et RMI.

Nexus. Nexus [89, 90, 85] est à l'origine un exécutif destiné au support du langage parallèle FORTRAN M [92], puis utilisé par d'autres langages parallèles comme Compositional C++ [59] et HPC++ [115], et enfin a été utilisé en tant que plate-forme générique principalement dans Globus [32]. Nexus est un environnement qui intègre la gestion du *multi-threading* et des communications. Dès le départ, l'accent a été mis sur la portabilité et la haute performance. Nexus intègre des mécanismes de gestion de protocoles de communication multiples : il est capable de s'adapter aux réseaux disponibles et peut éventuellement utiliser plusieurs protocoles simultanément, parmi TCP, UDP, MPI, IBM MPL, Intel NX, SHMEM. Théoriquement, la portabilité est transparente : l'application n'a pas à savoir quel est le protocole réellement utilisé pour acheminer ses données. Nexus sélectionne automatiquement le protocole adéquat et présente une interface uniforme quel que soit le protocole sous-jacent. Nexus assure également la portabilité des communications par rapport aux architectures des machines.

Nexus est basé sur le paradigme de communication d'*invocation à distance*, appelée en l'occurrence *Remote Service Request* (RSR). L'abstraction proposée est appelé *pointeur global* qui est incarné sous la forme d'un *lien* présentant un point de départ (ou *startpoint*) et un point d'arrivée (ou *endpoint*). Un lien est une connexion point-à-point unidirectionnelle sur laquelle circulent des requêtes allant du startpoint vers le endpoint. Un endpoint est lié à son *contexte* (processus), un startpoint peut être transféré d'un contexte à l'autre. L'établissement des connexions est entièrement dynamique.

L'implémentation depuis le début jusqu'à la version 4 (probablement la dernière) souffre de problèmes congénitaux. Son interface de programmation repose sur des tampons réutilisables ce qui semble problématique pour réaliser une implémentation "zéro-copie" ; ceci pénalise lourdement les performances. De plus, la sélection automatique du protocole réseau manque cruellement de flexibilité. L'ordre de préférence est codé en dur dans le code, et les canaux de communication doivent être ouverts dès l'initialisation, la dynamicité n'étant que "simulée" au-dessus de canaux fixes. En pratique, il n'y a pas de transparence d'utilisation puisque pour obtenir une répartition des liens Nexus sur les liens physiques de façon à obtenir une concurrence correcte, il est nécessaire d'intervenir manuellement dans la sélection de protocole. Nexus 4 est la base de MPICH-G, exécutif principal utilisé sur Globus 1. Depuis Globus 2.0, MPICH-G2 utilise Globus I/O, mince couche de portabilité, beaucoup plus rudimentaire mais plus léger ; Nexus est sur le déclin, victime de sa lourdeur, de ses médiocres performances, de sa complexité d'utilisation et de son modèle inadapté à l'utilisation que l'on a voulu en faire. Nexus est retiré de Globus depuis Globus 3.0.

Ces différents environnements génériques de communication sont destinés à être utilisés par divers exécutifs — des exécutifs basés sur le passage de messages, des mémoires virtuellement partagées, ou des langages parallèles —, qui sont décrits dans les sections suivantes.

2.2.3.2 Exécutifs pour le parallélisme à passage de messages

Nous présentons ici les principaux exécutifs et standards d'interface applicative utilisés pour le parallélisme à passage de messages.

PVM. Le projet PVM [172, 93] a débuté en 1989 à l'initiative de Vaidy Sunderam de l'université d'Emory (Atlanta, USA) et Al Geist du laboratoire national de Oak Ridge (Tennessee, USA) pour aider à la programmation d'applications distribuées en milieu hétérogène. À partir de 1991 l'audience de PVM s'est élargie sous l'impulsion de Jack Dongarra de l'université du Tennessee (Knoxville, USA).

PVM est un intergiciel pour le calcul parallèle distribué. PVM est basé sur le concept de *machine parallèle virtuelle* (PVM = *Parallel Virtual Machine*). Une machine virtuelle parallèle est un ensemble hétérogène et dynamique d'hôtes reliés par des réseaux, ensemble sur lequel la vue fournie par PVM est l'illusion d'un unique calculateur parallèle. Les communications entre les différentes tâches se font par échange de messages, en point à point ou diffusion.

Après bientôt 15 ans d'existence, PVM est toujours actif. La version 3.4 actuelle supporte de nombreuses architectures du simple PC jusqu'aux supercalculateurs, sous les variantes d'Unix ou sous Windows, pour la programmation en C, C++ et FORTRAN. PVM a été source d'inspiration pour de nombreux projets par la suite, et tout particulièrement MPI.

MPI. Au début des années 1990, nombreuses étaient les machines parallèles qui se programmaient d'une façon spécifique. Aucune portabilité n'était assurée : chaque application devait être adaptée à chaque type de machine. Pourtant, la plupart des bibliothèques de communication à échange de messages partageaient des notions similaires et auraient pu partager la même API. Partant de ce constat, un consortium d'universitaires et d'industriels, dirigé par Jack Dongarra et David Walker, décida de concevoir une interface standard pour le passage de messages sur les machines parallèles. En 1994, le MPI *forum* publia la spécification de MPI 1.0 [79], la première version de MPI (*Message Passing Interface*).

MPI est un standard d'interface de programmation pour passage de messages dont il existe de nombreuses implémentations. MPI est basé sur la notion de *communicateur*. Un communicateur est un

ensemble statique de nœuds de calcul reliés par un réseau. MPI assure les communications à l'intérieur d'un communicateur. Les communications prennent la forme d'échange de messages typés en point-à-point ; il propose également des opération collectives (diffusion, dispersion/regroupement, réduction). La version MPI 2.0 apporte notablement la dynamicité (*ie.* la possibilité d'ajouter ou retirer des nœuds en cours de session), des extensions telles que MPI I/O pour réaliser des entrées/sorties parallèles sur disques, des opérations d'écriture à distance (*one-sided operations*), et une prise en compte du *multi-threading*.

Le standard MPI a été massivement adopté par la communauté du calcul parallèle. L'immense majorité des constructeurs de machines parallèles ou de réseaux haute performance fournit une version de MPI pour utiliser leurs machines, même si les bibliothèques spécifiques perdurent. Il est courant que l'interface pour les applications soit MPI avec une implémentation basée sur une bibliothèque spécifique à plus bas niveau. En plus de celles fournies par les constructeurs, de nombreuses implémentations de MPI ont vu le jour, telles que MPICH [100] dérivé de la plate-forme de communication CHAMELEON [103], MPI-FM basé sur *Fast Messages*, ou encore LAM MPI pour les grappes basées sur Ethernet ou Myrinet. La plupart de ces implémentations sont basées sur la version MPI 1.2 ; MPI 2 peine à se généraliser.

Harness. Le projet *Harness* [134, 135, 125] (*Heterogeneous Adaptable Reconfigurable Networked Systems*) est l'héritier de PVM [93]. Il s'agit d'un projet conjoint entre l'université d'Emory (Atlanta, Georgie, USA), Oak Ridge National Laboratory (Tennessee, USA) et l'université du Tennessee (Knoxville, Tennessee, USA), dirigé par Vaidy Sunderam, Al Geist et Jack Dongarra.

Harness est un intergiciel pour le calcul parallèle à mémoire distribuée basé sur une approche à composants. *Harness* repose sur le concept de DVM (*Distributed Virtual Machine*), des machines virtuelles Java distribuées interconnectées. La DVM forme une plate-forme de base dans laquelle l'utilisateur charge dynamiquement à distance ses codes de calcul et des exécutifs, encapsulés dans des composants. Le point fort de *Harness* est son *adaptabilité* : tout étant dynamique (chargement des codes, participation d'une JVM à une DVM, etc.), le système peut être reconfiguré à la volée par l'utilisateur. La granularité de reconfiguration est cependant relativement grosse puisque chaque exécutif n'est constitué que d'un seul composant. Les exécutifs disponibles en *plugin* pour *Harness* sont PVM et MPI, ainsi que RMI Java qui est intégré à la JVM. Les exécutifs PVM et MPI peuvent être déployés sur les réseaux VIA en plus de TCP/IP grâce à un *plugin* spécifique. La performance obtenue [125] par *Harness* est conforme à ce que l'on peut attendre d'un système entièrement réalisé en Java, en deçà des performances de l'implémentation originale de PVM en C sur TCP/IP. Même si la priorité de *Harness* est l'adaptabilité plutôt que la performance, notons tout de même que le manuel de l'utilisateur de *Harness* conseille très vivement de mettre à profit JNI [124] pour pouvoir utiliser des codes FORTRAN et C depuis l'environnement Java, sans quoi les performances risquent d'être très mauvaises !

2.2.3.3 Mémoire virtuellement partagée

La notion de mémoire virtuellement partagée [154] ("MVP, ou couramment "DSM" — *Distributed Shared Memory*) est l'illusion d'une mémoire partagée au-dessus de mémoires physiquement distribuées. Plusieurs processus distribués sur des machines différentes partagent, par des mécanismes logiciels, un espace d'adressage commun de la mémoire. Les spécifications sur la façon de gérer les différentes copies d'une même donnée sont données par un *modèle de cohérence*, implémenté par un *protocole de cohérence*. Ce concept à été formalisé [127] en 1989 par Kai Li et Paul Hudak.

De nombreuses implémentations de DSM ont vu le jour. *TreadMarks* [40] a été développée à l'université de Rice (Houston, Texas) et est actuellement commercialisée. DSM-PM^2 [42] a été développée par Gabriel Antoniu au LIP (Lyon) et repose sur PM^2 ; sa spécificité est la prise en compte du *multi-threading*. MOME est développée par Yvon Jégou dans le projet PARIS à l'IRISA.

2.2.3.4 Langages parallèles

Les langages parallèles permettent de prendre en charge le parallélisme directement dans le langage plutôt que de gérer explicitement des échanges de messages ou des *threads*. Le programmeur donnes des indications sur la parallélisation dans son code ; le compilateur génère le code correspondant. La plupart des langages parallèles sont dérivés de langages séquentiels existants tels que C++ ou FORTRAN, et sont destinés à générer du code pour architectures parallèles à mémoire partagée.

Deux approches principales existent : les langages à parallélisme de tâches et les langages à parallélisme de données. Dans un langage à parallélisme de tâches, le programmeur donne des indications sur les parties qui peuvent être lancées en parallèle. Ces parties peuvent être des appels de fonctions lancés de façon asynchrone — c'est l'approche proposée par *Cilk* [50] du MIT et par *Athapascan* [57] développé au laboratoire ID-IMAG de Grenoble. La parallélisation peut concerner des schémas plus sophistiqués, en particulier les boucles, comme proposé dans le standard *OpenMP* [51]. Dans un langage à parallélisme de données, le parallélisme est dirigé par les données et non plus par le contrôle ; le programmeur spécifie la façon de découper les données pour les distribuer entre les *threads*. Les structures de base sont presqu'exclusivement des vecteurs et tableaux, les découpages sont souvent basés sur des blocs. C'est ensuite au compilateur de générer le code adapté, à prendre en charge la parallélisation, et à répartir ces tâches sur les différents processeurs. C'est notamment l'approche retenue par HPF [78] (*High Performance Fortran*), à l'origine développé à l'université de Rice (Houston, Texas).

Ces approches relèguent une partie de la complexité dans le compilateur. À l'exécution, du point de vue du système, le code généré se comporte comme un code *multi-thread* classique accompagné parfois d'une bibliothèque de support.

Cilk a été utilisé notamment pour des programmes d'échecs vainqueurs de plusieurs compétitions. HPF, qui présente sans conteste l'approche la plus riche et sans doute trop ambitieuse, est délaissé, ayant peu convaincu les utilisateurs potentiels à cause d'un manque d'implémentations complètes et efficaces. Le standard émergent est *OpenMP*, qui dans une certaine mesure peut être vu comme une version simplifiée de HPF. Il a été conçu pour des architectures à mémoire partagée, mais des travaux visent à exécuter des codes *OpenMP* sur des architectures parallèles distribuées *via* l'utilisation de DSM.

2.2.4 Conclusion sur les systèmes parallèles

Nous avons présenté dans cette section le calcul parallèle. Nous avons vu que les systèmes parallèles sont des systèmes concurrents où la priorité est donnée à la performance. L'architecture des machines, les modèles de programmation, et les infrastructures logicielles vont dans ce sens : toute complexité introduite est dans le but d'améliorer la performance.

Le parallélisme repose sur des hypothèses plus restrictives que les systèmes concurrents au sens large. Ces simplifications — gestion minimale de la sécurité, simplification de la topologie en un ensemble vu comme logiquement "à plat", architecture homogène — sont faites dans le but de permettre certaines optimisations pour obtenir de meilleures performances. Ceci a mené à la conception de machines dédiées, taillées sur mesure pour le parallélisme. La tendance actuelle des grappes prend en compte un autre facteur, à savoir le prix, en proposant de construire des ensembles dédiés au parallélisme à partir d'éléments standard bon marché.

Il ressort de l'étude des infrastructures logicielles qu'au niveau applicatif, la programmation repose sur des standards maintenant bien établis comme MPI pour le passage de messages, et OpenMP tend à devenir le standard en mémoire partagée. Ceci contraste avec la constellation de plates-formes génériques pour lesquelles il n'existe aucun standard.

2.3 Les systèmes répartis

Cette section est consacré aux *systèmes répartis*. Tout comme les systèmes parallèles, les systèmes répartis sont un cas particulier de système concurrent. Le but premier des systèmes répartis est la

répartition géographique des tâches. Nous posons la définition suivante qui sera celle de référence pour la suite de ce document :

Définition 2.17 : ***système réparti*** — Un système réparti est un système concurrent où la répartition géographique des tâches est l'aspect prépondérant.

Le système constitué d'un serveur *web* et de ses client est un exemple de système réparti. Il s'agit d'un système concurrent d'échange d'information, dont la base même de la conception est la répartition géographique. En effet, si le serveur *web* et ses clients devaient se trouver physiquement sur la même machine, la notion de serveur *web* perdrait de son intérêt ! Un système réparti est donc nécessairement distribué. Il est envisageable dans certains cas d'étendre cette définition à des systèmes distribués qui ne sont pas concurrents. La nécessité de répartition géographique vient de diverses contraintes :

– données produites ou consommées à des endroits particulier : par exemple, les données générées par un capteur particulier sont disponibles à l'emplacement du capteur ; il peut être nécessaire de les déplacer pour les traiter.
– emplacement des ressources de calcul : le choix est effectué selon la puissance des machines, la disponibilité, le prix, l'adéquation avec le problème (ex.: machine vectorielle, DSP spécialisés, etc.)
– codes de calcul : certains codes ne sont disponibles qu'à un endroit particulier et ne peuvent pas être diffusés pour cause de licence, d'incompatibilité, ou de clause de confidentialité.
– utilisateur : les interfaces homme-machine sont nécessairement localisées au même endroit que l'utilisateur ! Ceci inclut en particulier tous les dispositifs de visualisation.

Dans cette section, nous nous intéressons d'abord aux architectures réparties, puis nous décrivons les modèles de programmation du réparti, en enfin nous effectuons un tour d'horizon des principales infrastructures logicielles utilisées en calcul réparti.

2.3.1 Architectures réparties

Il n'existe pas à proprement parler d'*architectures réparties* par conception, car les systèmes répartis sont intrinsèquement un assemblage hétérogène de diverses ressources. La caractéristique principale des systèmes répartis est l'hétérogénéité : hétérogénéité des architectures, des systèmes d'exploitation, des logiciels, des réseaux, ou encore des politiques d'administration. Les systèmes répartis sont soumis à des problématiques contradictoires : la connectivité et l'interopérabilité d'une part, et la sécurité d'autre part.

La connectivité est la possibilité de se connecter à une machine distante, avec des intermédiaires *transparents*. Les intermédiaires assurent le routage, mais du point de vue de l'utilisateur, le système complet se comporte comme si la connexion vers la machine distante était directe. La connectivité est assurée par des protocoles chargés d'interconnecter plusieurs réseaux, éventuellement basés sur des technologies physiques différentes. Le protocole d'interconnexion le plus répandu est IP [150]. Les machines étant différentes, avec des systèmes et des logiciels différents, l'interopérabilité est assurée par une langue commune "sur le fil", spécifiée par des standards.

À ces aspects d'ouverture s'oppose la problématique de la sécurité. La sécurité a pour but de se protéger des intrusions et interceptions, donc apporte inévitablement des limitations de l'ouverture. Il en résulte l'omniprésence de pare-feux, machines servant de garde barrière pour l'entrée sur un site, filtrant les messages, brisant volontairement la connectivité. Par conséquent, la connectivité entre toutes les machines d'un système réparti sur plusieurs sites est souvent incomplète ; la connexion entre deux sites peut alors tout de même se faire grâce à une passerelle qui assure le relais. Une autre approche, connue sous le nom de IP*sec* [118] prend en compte la sécurité dès le niveau du protocole IP.

Un système réparti est constitué de machines diverses, stations de travail ou machines de calcul reliées par des réseaux LAN et/ou WAN, voire des machines éparpillées sur *internet*. Nous distinguons les variantes suivantes :

– Un système réparti local, ou *intranet*, emprunte un LAN et reste à l'intérieur d'un domaine d'administration. Typiquement, il regroupe des stations de travail et/ou des serveurs sur un unique

site, à l'abri des préoccupations de sécurité.
- Un système réparti à large échelle emprunte des WAN et des LAN et se trouve sur plusieurs domaines d'administration. Typiquement, il regroupe des machines de plusieurs institutions et/ou entreprises qui collaborent ou mettent en commun leurs infrastructures.
- Un système réparti utilisant un grand nombre de machines éparpillées sur *internet*, toutes les machines ayant un rôle équivalent — à la fois client et serveur —, est qualifié de système *pair-à-pair*. Les utilisations actuelles sont le partage de données et la mise à disposition du surplus de temps de calcul des machines des particuliers.

La notion de *système réparti* est souvent un concept abstrait. Le système lui-même, en tant qu'entité composite, a une existence virtuelle. La plupart du temps, des machines entrent et sortent du système réparti de façon dynamique. La vue de chaque nœud se restreint uniquement à une vision locale, aux voisins immédiats (*ie.* ceux avec qui on dispose d'une connexion directe). Alors qu'en parallélisme la notion topologique de base est un *communicateur*, à l'intérieur duquel tous les nœuds sont connectés à tous les autres, en réparti la topologie de base est un *lien* qui relie deux nœuds. Le couplage entre l'ensemble des nœuds du système est faible et la participation des nœuds au système est dynamique. La dynamicité repose sur la capacité de nommer des nœuds que l'on ne connaît pas *a priori*. Tout système réparti repose donc sur un mécanisme d'*adressage* qui permet de désigner des nœuds arbitraires.

2.3.2 Modèles de programmation répartie

Dans cette section, nous décrivons les modèles de programmation utilisés pour programmer les systèmes répartis. Dans la programmation de tels systèmes, l'accent est souvent mis sur la robustesse et l'interopérabilité, parfois au détriment de la performance. La programmation en réparti se fait essentiellement selon trois modèles :

Client-serveur — Le modèle de programmation le plus fréquent en réparti est le client-serveur. C'est un modèle asymétrique où un nœud joue le rôle de serveur et un autre nœud joue le rôle de client. Le serveur attend les connexions venant des clients. Une spécialisation du client-serveur est l'appel de service à distance. Dans ce cas, quand un client se connecte au serveur, il envoie sa requête accompagnée de ses arguments, ce qui déclenche éventuellement une action sur le serveur, puis le client reçoit le résultat de sa requête. Le client-serveur est le mode de fonctionnement typique des serveurs *web*, des RPC (*Remote Procedure Call* — appel de procédure à distance), des RMI (*Remote Method Invocation* — appel de méthode à distance, le serveur étant alors un objet), ou des *Web Services*.

Fédérations — Dans une fédération, une entité centrale, appelée *fédérateur* régit le déroulement du calcul et toutes les interaction à l'intérieur du système. Des nœuds "clients" appelés *membres*, pouvant rejoindre ou quitter la fédération en se connectant ou déconnectant du fédérateur, apportent leur contribution au calcul. Le fédérateur et les membres de la fédération communiquent en échangeant des messages. Ce modèle est par exemple utilisé dans l'architecture de simulateur HLA [109] du DMSO (*Defense Modeling and Simulation Office*), ou dans des programmes de calcul sur *internet* tels que SETI@home [30] ou Folding@home [23].

Pair-à-pair — Un système pair-à-pair (abrégé en "*p2p*") est un système réparti où tous les nœuds, appelés alors "*pairs*" sont logiquement équivalents. Tous les pairs jouent à la fois le rôle de client et de serveur ; ils peuvent se connecter de façon arbitraire les uns aux autres (d'où le nom "pair à pair") et pas uniquement à un fédérateur. Les exemples de *p2p* sont majoritairement des services de partage de données. Il existe également des exemples de *p2p* pour le calcul tels que *XtremWeb* [96]. Les caractéristiques principales de la programmation sur *p2p* sont la grande volatilité des nœuds de calcul — ils peuvent se déconnecter à tout moment sans prévenir —, et l'hétérogénéité des performances, aussi bien pour les machines que pour les liens réseaux.

2.3.3 Infrastructures logicielles pour le calcul réparti

Dans cette section, nous réalisons un tour d'horizon des infrastructures logicielles utilisées pour programmer les systèmes répartis. Cette section se subdivise en trois parties, par niveau d'abstraction

croissant : les protocoles et interfaces système, les plates-formes génériques d'accès au réseau, et les intergiciels.

2.3.3.1 Protocoles et interfaces systèmes

Nous décrivons les protocoles et interfaces du système d'exploitation utilisées pour les réseaux des systèmes répartis. Nous présentons d'abord une pile de protocole "habituelle" composée de bas en haut de IP, TCP, *sockets* ou XTI, HTTP, puis nous présentons x-kernel qui représente une alternative à la pile complète.

Les protocoles *internet*. Le protocole standard qui permet l'interconnexion de réseaux à large échelle est sans conteste le *protocole internet* [150], plus connu sous le sigle "IP". Il s'agit d'un protocole de niveau "réseau" dans la hiérarchie OSI [37], c'est-à-dire qu'il est chargé d'acheminer des paquets à travers une interconnexion de réseaux pouvant reposer sur des technologies différentes. C'est dans cet esprit que le protocole IP a été conçu. Ce protocole amène également un système standard d'adressage des machines.

Au-dessus du protocole IP, deux principaux protocoles de niveau "transport" sont construits : UDP (*User Datagram Protocol*) et TCP (*Transfer Control Protocol*). UDP est presque une projection directe de IP au niveau transport, en ajoutant seulement le multiplexage permettant à plusieurs applications de partager un réseau ; UDP est ainsi basé sur des paquet (appelés *datagrammes*), il n'est pas fiable — des datagrammes peuvent être perdus, arriver dans le désordre ou être dupliqués —, sans contrôle de flux — *ie.* il émet sans savoir si le récepteur est prêt à recevoir. TCP [151] est un autre protocole construit au-dessus de IP ; il ajoute de nombreux services. TCP est basé sur un flux continu de données, faisant disparaître les frontières de paquets, fiable, et met en œuvre des mécanismes de contrôle de flux et de congestion. TCP/IP est le principal protocole utilisé sur *internet*. Dans la suite de ce document, nous supposons qu'un hôte "*internet*" met en œuvre IP, TCP et UDP conformément au standard décrit par la RFC 1122 [54] définissant un hôte *internet*.

Les protocoles *internet* sont en constante évolution. Le protocole IP est en cours de transition de IP *v4* vers IP *v6*, qui élargit notamment l'espace d'adressage de 32 bits à 128 bits et prend en compte la qualité de service. Au niveau supérieur, TCP et UDP se voient complétés par le nouveau protocole SCTP [171] qui apporte une plus grande flexibilité dans la fiabilisation, est conçu dans l'optique des réseaux mobiles, et comprend les dernières avancées en matière de contrôle de congestion et de sécurité.

Interfaces de programmation : *sockets* BSD, *System V* TLI et Posix. Les protocoles *internet* présentés ci-dessus sont programmés à l'aide d'une API du système. Historiquement, la première interface standard pour programmer TCP/IP est l'API *socket* Berkeley, dont la première mouture a été introduite en 1983 avec 4.2 BSD. Dans la famille des Unix "*System V*", l'Unix SVR3 d'AT&T se dotait d'une interface appelée TLI (*Transport Layer Interface*) en 1985, fruit d'un portage des *streams* [158] de Dennis Ritchie. Les deux systèmes ont coexisté, les *sockets* BSD bénéficiant d'une plus large diffusion, l'interface TLI offrant plus de fonctionnalités. L'API *socket* a évolué pour devenir telle que nous la connaissons actuellement dans 4.4 BSD. Dans le même temps, le consortium *X/Open* reprenait TLI et le faisait évoluer en XTI [104] (*X/Open Transport Interface*) pour l'intégrer dans les spécifications Unix 98 de l'*Open Group*.

En 2000, l'IEEE finalise enfin le standard définissant les interfaces de communication indépendantes du protocole dans la norme IEEE 1003.1g [38] (connue aussi sous le nom "Posix.1g") ; cette norme contient à la fois les API issues de BSD et de *System V* : *sockets* et XTI. Les extensions temps-réel IEEE 1003.1b [36] apportent encore une interface supplémentaire, à savoir Posix AIO (*Asynchronous I/O*) pour gérer les communications asynchrones efficacement. Le standard Posix AIO étant très vague et décrié par certains constructeurs pour son inadéquation aux besoins, certaines versions ne sont volontairement pas conformes, comme Linux AIO ou SGI KAIO. Pour ajouter encore à la multitude d'interfaces, notons l'apparition récente de mécanismes destinés aux entrées/sorties à haut niveau de concurrence, par exemple la scrutation temps-réel `/dev/epoll` [128] sous Linux et les `kqueue` [126] dans FreeBSD.

Aussi bien les *sockets* que XTI reposent sur le concept de *descripteurs*, des entités manipulées par un programme et permettant les interactions avec la couche réseau du système. Les deux interfaces sont également capables de manipuler plusieurs protocoles, avec des propriétés qui dépendent du protocole utilisé.

Nous observons une diversité des API. Toutefois, les diverses API servent à accéder aux mêmes protocoles ; tous les systèmes restent interopérables quand ils manipulent le même protocole. Un client qui utilise des *sockets* peut par exemple se connecter à un serveur basé sur XTI si tous les deux utilisent TCP/IP. De plus, les *sockets* BSD tendent à s'imposer comme le standard "de fait" et sont disponibles dans la plupart des systèmes d'exploitation.

HTTP. HTTP [72] (*Hypertext Transfer Protocol*) est un protocole de niveau utilisateur utilisé pour le transfert de données de types variés. Il est basés sur la notion de client/serveur : un client envoie une requête contenant une URL (*Uniform Resource Location*) qui identifie l'adresse et le nom de l'objet demandé ; le serveur répond en fournissant l'objet demandé. À l'origine HTTP était utilisé uniquement pour transférer des pages *web*. Depuis, son usage se généralise à une multitude d'autres applications, détrônant les applications où FTP était utilisé, et est même maintenant utilisé pour des RPC (voir les *Web Services*, dans la section "intergiciels pour le réparti").

x-kernel. x-kernel est une alternative aux piles habituelles *sockets*/TCP/IP présentées ci-dessus, et propose une approche globale pour implémenter des piles complètes de protocoles. L'idée de départ de x-kernel est la question : "quel est le bon nombre de couches ?" dans une pile de protocoles réseau. À l'inverse de OSI [37] qui répond "7", la réponse proposée par Hutchinson et Peterson [106, 144] de l'université d'Arizona est : "ça dépend !". Les paramètres sont multiples : complexité du protocole, optimisation, réseaux utilisés, etc. Le modèle OSI, trop rigide, n'est pas toujours l'optimal. Pour parvenir à cette flexibilité, ils proposent une architecture basée sur des *micro-protocoles* (composants) qui implémentent une fonctionnalité précise d'un protocole. L'assemblage des divers micro-protocoles au sein de x-kernel permet de construire des protocoles de plus haut niveau. Le principe essentiel de x-kernel est la composition des micro-protocoles, composition basée sur les notions de :

encapsulation — un micro-protocole est une entité indépendante qui ne fait pas de supposition sur l'environnement extérieur, qui ne dépend pas de l'état d'autres entités.

standardisation des interface — pour permettre la composition arbitraire de tous les micro-protocoles sur tous les autres.

sous-spécification — à l'inverse de la tendance répandue à *sur-spécifier*. Par exemple UDP n'est pas intrinsèquement lié à IP alors que son interface d'adressage le lie artificiellement à IP ; en ce sens, UDP est sur-spécifié.

instanciation multiple — chaque micro-protocole peut être instancié plusieurs fois

ré-utilisabilité — les micro-protocoles sont suffisamment simples pour être réutilisés dans d'autres contextes que ceux dans lequel ils ont été écrits. Le cas échéant, leur modification est simple du fait de leur petite taille.

L'exemple typique est le protocole Sprite-RPC (un protocole de RPC léger) qui est décomposé en les micro-protocoles (de bas en haut) : Ethernet, IP, fiabilisation (BLAST), requête/réponse (CHANNEL), démultiplexage (SELECT). Dans ce cas, les micro-protocoles Ethernet, IP et fiabilisation sont intuitivement réutilisables pour construire d'autres protocoles, requête/réponse également (serveur *web*, par exemple).

Pour ajouter à la flexibilité et permettre réellement un nombre de couches variable d'un protocole à l'autre, mais aussi d'un message à l'autre, x-kernel propose la notion de *protocole virtuel*. Un protocole virtuel est une entité qui peut être composée avec d'autres micro-protocoles mais ne réalise comme action que le *choix* du micro-protocole à utiliser. On peut les voir comme des outils d'assemblage dynamique. Dans l'exemple précédent, il est par exemple possible de connecter directement le micro-protocole de fiabilisation à Ethernet sans intercaler IP dans le cas où le message ne sort pas d'un réseau Ethernet. Une telle décision est prise par un protocole virtuel.

x-kernel a été implémenté en tant que système d'exploitation pour que ses principes soient appliqués sur toute la chaîne, sans dépendre par exemple d'une implémentation TCP/IP médiocre d'un constructeur. Parmi les protocoles implémenté, on trouve le multicast, le RPC, les flux continus. x-kernel en tant que système d'exploitation à part entière a rapidement disparu, mais a ensuite été intégré au micro-noyau Mach version 3.2 de l'*Open Software Foundation*.

À tous les niveaux (IP, TCP, *sockets*, etc.), le but de ces interfaces systèmes et protocoles est d'exploiter au mieux les infrastructures réseau en s'affranchissant de l'hétérogénéité. À chaque niveau, l'interopérabilité guide de nombreux choix.

2.3.3.2 Bibliothèques d'accès au réseau

Tout comme en distribué, il existe en réparti des plates-formes génériques — au sens de la définition 2.12 pour factoriser les parties de code réutilisables d'un exécutif à l'autre. Les schémas de programmation pour gérer les communications en réparti sont en effet souvent les mêmes : construire un message et l'envoyer pour un client, attendre les requêtes de clients et les traiter pour un serveur.

Plusieurs propositions ont vu le jour pour factoriser le code nécessaire à ces opérations : l'abstraction de haut niveau selon les principes décrits à la section 2.1.3.2, les boucles d'évènements — attendre les évènements en provenance du réseau et de les traiter —, et les environnements à composant selon les principes décrits à la section 2.1.3.3. Nous décrivons ici successivement ces trois approches de plate-forme générique dédiée au réparti, basée sur les protocoles décrits à la section précédente.

Abstraction pour le réparti : ADAPTIVE. ADAPTIVE [164, 165] (connu aussi sous le nom ACE ™ — ADAPTIVE *Communication Environment* —, "ADAPTIVE" étant lui-même l'acronyme de *A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment*) est un projet initié par Douglas Schmidt, à l'université de Washington à St. Louis. ADAPTIVE est un environnement de développement portable pour les applications réparties ; ADAPTIVE apporte la portabilité par rapport aux communications, aux *threads*, à la gestion de la mémoire, et fournit des abstractions de haut niveau en C++ pour programmer les différentes ressources. ADAPTIVE a été utilisé principalement pour l'implémentation CORBA temps-réel TAO [122], et pour le serveur *web* JAWS.

Boucles de gestion d'évènements réseaux. De nombreux serveurs sont basés sur la même structure : dans une boucle attendre les requêtes de clients et les traiter une par une. Des environnements spécifiques ont été développés pour gérer ces opérations. Par exemple, la *GLib Main Event Loop* (également connue sous le nom `GMainLoop`) est l'initiative du projet Gnome [24]. Elle offre une infrastructure générale pour gérer la boucle principale d'évènement d'un programme, y compris les entrées/sorties pour le réseau.

Beaucoup de plates-formes de gestion d'évènements réseaux sont spécialisées dans la gestion des hauts niveaux de concurrence. Le point de départ est le problème "C10K" [117] (10 000 clients) : une machine à 1 GHz dotée de 2 Go de mémoire devrait être capable de gérer 10 000 clients simultanément ; ceci représente 100 000 cycles de processeur et 50 Ko de mémoire par client, ce qui devrait être largement suffisant. Pourtant, à moins d'utiliser des méthodes très spécifiques, la plupart des systèmes n'y parviennent pas : le temps d'accès au réseau devient proportionnel au nombre de connexions ouvertes, et le serveur n'est pas capable de soutenir la charge. Niels Provos, de l'université du Michigan, propose `libevent` [155], une bibliothèque d'accès au réseau capable de soutenir le haut niveau de concurrence en utilisant les méthodes `kqueue` [126] de BSD et `/dev/epoll` [128] de Linux. Dan Egnor propose `liboop` [71], solution similaire à `libevent` mais basée sur une interface abstraite qui généralise le concept de boucle d'évènements. Ces deux bibliothèques sont utilisées dans de nombreux serveurs devant à la fois être portables et utiliser les mécanismes de scrutation les plus récents pour supporter de hauts niveaux de concurrence.

Plates-formes à composants. L'adaptation à un environnement hétérogène est un contexte de choix pour les approches basées sur des composants. En effet, l'assemblage d'éléments "à la carte" permet

une adaptation aux protocoles proposés et interfaces demandées. Bobby Krupczak *et al.* du Georgia-Tech proposent [121] un environnement construit avec des composants permettant de construire une plate-forme avec uniquement les composants souhaités selon le protocole et l'interface voulue.

Fischer et Ercegovac de l'université de Californie à Los Angeles proposent [73] un environnement de communication appelé DAT basé sur une interconnexion de composants réutilisables. L'approche est basée sur trois abstractions : le *message*, le *protocole*, et le *graphe*. Les protocoles (composants) dérivent de la classe *protocole* ; ils implémentent une fonctionnalité de communication bien déterminée, par exemple un service de datagramme ou une agrégation de plusieurs flux. Le *graphe* est une description d'un assemblage particulier de *protocoles*. Les connexions entre composants sont réalisées par l'environnement en suivant le graphe. Le *message* est l'entité qui transporte les données et qui circule d'un protocole à l'autre. L'environnement est implémenté en C++. Il utilise les primitives de communications fournies par le système, telles que les *sockets*. L'approche proposée semble flexible et présente des caractéristiques de dynamicité et d'encapsulation des composants. Cependant, la flexibilité et l'extensibilité restent à prouver, les résultats publiés [73] ne faisant la démonstration que sur un protocole à datagrammes construit au-dessus de TCP/IP ! De plus, les performances publiées sont relativement médiocres.

En conclusion, les plates-formes à composants pour gérer le niveau générique en réparti semble une approche prometteuse. Toutefois, aucune proposition concrète n'obtient de de résultats satisfaisants en allant au-delà d'une simple étude préliminaire.

2.3.3.3 Intergiciels pour le réparti

Nous décrivons ici les principaux intergiciels utilisés en calcul réparti, en particulier ceux basés sur les modèles RPC et fédérations. Ces intergiciels sont utilisés par les applications et utilisent les protocoles ou les environnements génériques décrits ci-dessus.

RPC : *Sun* RPC et DCE. Les mécanismes de RPC les plus répandus sont celui introduit par *Sun Microsystems* puis standardisé [170] par l'IETF, et DCE [105] de l'*Open Group*.

En RPC, les appels sont traité par un processus qui attend les requêtes des clients. Les procédures appelées sont identifiée par un entier dont la correspondance avec un service réel est assurée par une table de procédures standard. Le transport n'est pas fixé ; couramment, UDP et TCP sont utilisés. Les paramètres des procédures sont encodés en format XDR pour assurer l'interopérabilité en environnement hétérogène.

Les RPC *Sun* sont utilisés couramment sur de nombreux systèmes, dont les utilisations les plus notables sont NFS (*Network File System*) et NIS (*Network Information Service*), deux services qui permettent de partager respectivement des disques et des tables d'informations telles que la table des mots de passe ou des machines d'un réseau. Les RPC sont en général limités à un usage local sur un seul domaine d'administration.

L'architecture CORBA. CORBA [146] est un standard d'architecture d'intergiciel à objets distribués. C'est l'acronyme de *Common Object Request Broker Architecture* — architecture commune pour courtier de requêtes à objets. CORBA est une norme définie par l'OMG [33] (*Object Management Group*), consortium ouvert qui réunit des industriels et des universitaires et qui a pour but de normaliser les technologies orientées objet. Il existe de nombreuses implémentations de la norme CORBA, dont certaines sont des logiciels libres. La dernière version est CORBA 3.0.2. CORBA étant une norme, l'OMG fait passer aux différentes implémentations un test de conformité à la norme. Une implémentation qui passe le test avec succès est dite "certifiée".

Jusqu'aux version CORBA 2.x, CORBA définit une architecture d'objets distribués. À partir de la version CORBA 3.0, il est ajouté une architecture de composants logiciels [173] appelée CCM (CORBA *Component Model*) ; à l'exécution, ces composants reposent toutefois sur des objets CORBA à la façon de CORBA 2.x. Il est d'ailleurs toujours possible de programmer en objets avec CORBA 3.0. L'architecture d'objets ou composants définie par CORBA est indépendante du type de machine, du système

d'exploitation, et du langage de programmation utilisé. L'architecture CORBA est composée essentiellement de l'ORB (*Object Request Broker* — courtier de requêtes objet) qui est le cœur de l'architecture CORBA, et des objets pris en charge par l'ORB. L'ORB assure le transport des requêtes sur le réseau. Du point de vue du programmeur, n'importe quel objet CORBA est accédé de façon transparente, comme si c'était un objet local. L'ORB se charge d'intercepter les invocations de méthode, de localiser l'objet, d'acheminer par le réseau les invocations avec leurs paramètres et de transmettre les éventuelles valeurs de retour des méthodes. Les objets définis par le programmeur sont pris en charge par l'ORB. Les interfaces des objets et composants CORBA sont spécifiées à l'aide du langage IDL (*Interface Definition Language*) qui permet l'indépendance par rapport au langage d'implémentation.

Pour que différentes implémentations CORBA puissent communiquer entre elles, la norme spécifie un protocole de transport des requêtes. Ce protocole s'appelle GIOP (*General Inter-*ORB *Protocol*). Il spécifie un encodage des paramètres d'entrée et des valeurs de retour des méthodes, ainsi qu'un format de messages. L'incarnation de GIOP sur TCP/IP s'appelle IIOP (*Internet Inter-*ORB *Protocol*). S'il le juge utile, un ORB peut utiliser n'importe quel protocole pour les communications entre un client et un serveur. S'il s'affranchit de GIOP, on parle alors d'ESIOP (*Environment-Specific Inter-*ORB *Protocol*). Pour l'interopérabilité, il est cependant nécessaire que chaque objet puisse être aussi accédé par IIOP. Par souci de simplicité, la plupart des ORB utilisent exclusivement IIOP (et donc TCP/IP) pour toutes leurs communications.

Nous présentons ici quatre implémentations CORBA basées sur le langage C++, largement répandues, et qui sont disponibles gratuitement.

MICO [25], développé à l'université de Francfort (Allemagne), présente l'avantage d'être une implémentation complète OpenSource, respectant strictement la norme CORBA 2.3. Il n'est pas réputé être l'ORB le plus performant, mais ses performances sont toutefois honorables. MICO offre une implémentation CORBA 3.0 appelée MICOCCM.

TAO [163] (The ACE ORB) est développé à l'université de Washington (USA). TAO est basé sur la plate-forme générique ACE décrite ci-dessus qui fournit une abstraction des ressources et prend en charge la plupart des problèmes de portabilité. TAO est OpenSource, et est un ORB complet certifié CORBA 2.3 destiné principalement aux applications "temps réel".

ORBacus [27] est distribué par Iona ; c'est un ORB certifié CORBA 2.3. Il est gratuit pour une utilisation non commerciale. Il s'appelait auparavant OmniBroker.

OmniORB [45] était à l'origine développé aux laboratoires AT&T de Cambridge (Royaume-Uni), aujourd'hui maintenu par Apashphere Ltd. C'est un ORB OpenSource. Il est particulièrement simple et efficace. OmniORB 3 est certifié conforme aux spécifications CORBA 2.1. OmniORB 4 implémente un sous-ensemble de la norme CORBA 2.6.

Il existe de nombreuses autres implémentations CORBA pour presque tous les langages (Java, Python, Perl, C, Ada, C#, etc.) La seule exception notable est l'absence de projection officielle vers le langage FORTRAN, bien qu'une version non-reconnue par l'OMG existe.

***ZeroC* ICE.** ICE ™ (*Internet Communications Engine*) est un intergiciel conçu par la société *ZeroC*, fondée par Michi Henning, l'un des principaux contributeurs de la norme CORBA. Selon son créateur, ICE est le successeur de CORBA, corrigeant les erreurs qui avaient été faites par le passé et qui ne pouvaient pas être corrigées dans CORBA sous peine de briser la compatibilité. Les améliorations par rapport à CORBA concernent essentiellement une refonte complète du langage de description d'interface (appelé *Slice* dans ICE) et du système de typage, d'une simplification du mapping C++, et de l'intégration dès le départ de la gestion de la sécurité, de la persistance, et des messages asynchrones, choses incluses dans CORBA mais non prévues lors sa conception. À la différence de CORBA, ICE n'est régi par aucun standard ; il existe une seule implémentation, celle de *ZeroC* disponible gratuitement pour des utilisations non-commerciales. Cependant, son protocole ouvert rend possibles d'autres implémentations.

RMI Java. Quand *Sun Microsystems* a conçu son langage de programmation Java, il a intégré à partir de la version 1.1 un mécanisme de RPC— ou plutôt de RMI (*Remote Method Invocation*), puisqu'il s'agit

de méthodes d'objets. De façon similaire à CORBA, les objets accessibles à distance sont déclarés explicitement, créés, et référencés par un nom. Cependant, la description se fait directement en langage Java, et c'est la machine virtuelle Java qui joue le rôle de courtier d'objet, c'est-à-dire qui achemine les invocations de méthodes faites sur une référence vers l'implémentation réelle de l'objet. Les RMI Java sont limités aux interconnexions de programmes Java. Toutefois, très rapidement *Sun* a ajouté une version des RMI basée sur le protocole IIOP de CORBA, fournissant du même coup l'interopérabilité avec CORBA.

***Web Services* et SOAP.** Les *services web*, ou "*Web Services*" [58] sont au départ toute infrastructure logicielle fournissant un service accessible à distance par le *web*. Le principe a été formalisé [182] par le *W3C* sous la forme de quatre entités :

– un service de *transport* ; le plus répandu est HTTP [72] mais d'autres sont possibles, comme FTP ou SMTP.
– un service de *messages* ; ce point est sujet à débat pour savoir si le principe des *Web Services* est lié à XML ou non. En tout état de cause, les implémentations existantes basent leurs échanges de messages et l'interopérabilité sur XML [180]. Les protocoles d'échange de messages courants sont SOAP (*Simple Object Access Protocol*) [181] et XML-RPC, une version allégée de SOAP.
– un service de *description* pour décrire l'interface publique du service. Actuellement, ceci est effectué à l'aide du langage WSDL.
– un service de *découverte*, pour localiser les services (service de *pages jaunes*). Actuellement, ce service est fourni par UDDI.

Même si la définition des services *web* est large et flexible, dans la pratique la majeure partie des services *web* repose sur HTTP, SOAP, WSDL et UDDI. Toutefois, l'usage de XML qui semblait systématique devrait être remis en question ; par exemple, *Sun* étudie la possibilité d'utiliser plutôt le langage ASN.1 [113] pour améliorer les performances de transfert. Puisque nous nous intéressons essentiellement aux communications, en l'état actuel des choses, nous retiendrons que les services *web* reposent majoritairement sur SOAP.

***High Level Architecture* (HLA).** HLA est un standard d'architecture pour des *fédérations* d'objet dans le domaine des simulations. Par le passé, les simulations militaires du département de la défense américain étaient réalisées au cas par cas de façon spécifique, et étaient donc interopérables entre elles uniquement au prix de lourds développements d'adaptation, et guère réutilisables. En 1994, l'agence pour les projets de recherche avancés (DARPA) du département de la défense américain débute une initiative de standardisation pour les fédérations de simulateurs pour remédier aux problèmes d'interopérabilité de réutilisabilité des simulateurs. En 1996, le groupe chargé d'établir ce standard terminait son travail en publiant la spécification d'une architecture de fédération de simulateurs appelée HLA [109], standard adopté par l'IEEE en 2000.

HLA est une architecture qui définit des règles d'interactions, un modèle d'objets, et une spécification d'interface. La fédération est gérée par une entité appelée RTI (*Runtime Infrastructure*). Le DMSO (*Defense Modeling and Simulation Office*) fournit une implémentation de référence connue sous le nom RTI 1.3 NG.

2.3.4 Conclusion des systèmes répartis

Nous avons présenté dans cette section les systèmes répartis, qui sont des systèmes concurrents dont la caractéristique essentielle est la répartition géographique. Les systèmes répartis amènent de nombreuses problématiques à gérer : l'hétérogénéité, l'interopérabilité, la sécurité, la fiabilité, etc. Ce sont des systèmes intrinsèquement complexes. Les infrastructures logicielles que nous avons présentées vont dans le sens de la facilité de gestion de cette complexité. Par exemple, un grand effort est fait pour masquer l'hétérogénéité. Ainsi, au niveau des protocoles, IP a été conçu pour pouvoir emprunter de nombreux supports physiques, les API de communication *sockets* et XTI peuvent servir à programmer de nombreux protocoles. Les bibliothèques de communication telles qu'ADAPTIVE, `liboop` ou

`libevent` cherchent à utiliser la meilleure méthode spécifiquement pour chaque système tout en gardant la même interface. Au niveau des intergiciels, l'effort porte essentiellement sur la transparence de localisation pour masquer la répartition géographique. La majorité des systèmes de RPC (*Sun*, CORBA, ICE ou RMI Java) sont pensés pour que les invocations à distance soient aussi faciles que des invocations locales, de façon à offrir l'illusion au programmeur que le système est concurrent, certes, mais pas distribué.

2.4 Conclusion

Nous avons présenté dans ce chapitre une vue d'ensemble des systèmes concurrents, subdivisée en des définitions générales, puis une description des systèmes parallèles puis des systèmes répartis. Nous avons vu que les différents types de systèmes concurrents partagent beaucoup de leurs problématiques : plusieurs tâches fonctionnent en même temps, elles réalisent des communications, des infrastructures logicielles existent pour aider le programmeur à gérer les aspects distribué et concurrent.

Cependant, des approches radicalement différentes sont mises en œuvre par les systèmes parallèles et les systèmes répartis : les systèmes répartis sont constitués de ressources diverses, hétérogènes, que l'on cherche à faire interopérer, éventuellement avec d'autres machines extérieures, alors que l'approche du parallélisme se réserve le droit de faire certaines hypothèses particulières dans le seul but de se concentrer sur la haute performance ; ces hypothèses peuvent être l'homogénéité, la suppression de l'interopérabilité (système fermé), la sécurité supposée gérée à l'extérieur, et même la conception de machines spécialement dédiées au parallélisme.

Les systèmes parallèles et répartis ont donc beaucoup de problématiques en commun. Pourtant, les hypothèses de base, les buts et les priorités étant radicalement différents, les solutions apportées à ces problématiques sont différentes. On constate en pratique que les mondes du parallélisme et des systèmes répartis, malgré de nombreuses similitudes, interagissent très peu.

Chapitre 3

Les grilles de calcul

Sommaire

3.1 Notion de grille de calcul 33
3.1.1 Les grilles : origines et évolution 33
3.1.2 Architecture des grilles de calcul 35
3.1.3 Infrastructures logicielles pour les grilles 35
3.2 Voir une grille comme un calculateur parallèle 37
3.3 Voir une grille comme un système réparti 38
3.4 Nouvelles possibilités des grilles 40
3.5 Conclusion 42

Après la présentation des systèmes concurrents "classiques" au chapitre précédent, nous nous intéressons dans ce chapitre au domaine émergent des grilles de calcul. Les grilles de calcul combinent des aspect des systèmes parallèles et des systèmes répartis. Elles représentent un passage à large échelle, aussi bien au niveau applicatif qu'au niveau des ressources employées. Enfin, le point le plus important à nos yeux, les grilles permettent de dépasser les modèles "classiques" de systèmes concurrents où le modèle applicatif et le modèle du matériel sont étudiés l'un en fonction de l'autre ; à l'inverse, sur les grilles il est courant de découpler les topologies et paradigmes du niveau logiciel de ceux du niveau matériel.

Ce chapitre s'organise en quatre parties. Nous commençons par une présentation générale des grilles de calcul. Puis nous nous intéressons à la façon de les programmer : d'abord les programmer comme des systèmes parallèles, puis comme des systèmes répartis, et enfin les considérer comme un monde à part entière avec de nouvelles possibilités par rapport au parallélisme et au réparti.

3.1 Notion de grille de calcul

Les grilles de calcul sont un domaine en plein essor qui est le point de rencontre de personnes de tous horizons, chaque personne ayant sa propre vision des grilles. La définition de ce qu'est une grille de calcul est sujette à débat et suscite parfois de vives polémiques. Dans cette section, nous traçons un bref historique, nous définissons ce que nous entendons par *grille de calcul*, puis nous décrivons les principaux acteurs dans le domaine des solutions logicielles pour les programmer.

3.1.1 Les grilles : origines et évolution

L'émergence des réseaux longue distance à haut débit a permis d'imaginer d'utiliser des machines sur plusieurs sites pour exécuter des applications parallèles sur un grand nombre de nœuds. En 1995,

les États Unis lancent un réseau expérimental baptisé *I-Way* [67] (*Information Wide Area Year*), basé sur des liens ATM à 155 Mbit/s reliant onze centres de calcul à travers le pays. Une infrastructure logicielle appelée *I-Soft* [86] est mise au point pour gérer les lancements d'application et les transferts de données, en particulier pour les démonstrations de la conférence *Supercomputing 1995*. L'exécutif pour le parallélisme Nexus [89] développé au laboratoire national d'Argonne est alors adapté pour cet usage à grande échelle et une implémentation MPI est réalisée [82] sur Nexus pour une utilisation sur *I-Way*. Parallèlement, des travaux effectués à l'université de Virginie visent à exécuter sur le réseau *I-Way* des applications basées sur Legion [99], à l'origine exécutif pour le langage parallèle *Mentat* dont la portée a été élargie à d'autre langages et à plus large échelle. On employait alors le mot "*metacomputing*" pour désigner ces déploiement à grande échelle.

Très rapidement, le mot "*grid*" (en français : "grille") a supplanté l'appellation *metacomputing* qui est désormais réservée à quelques cas particuliers. La définition exacte d'une grille de calcul est encore floue. S'agissant d'un domaine relativement récent et encore en pleine ébullition, aucune définition précise universellement reconnue n'existe à l'heure actuelle ; l'idée intuitive de *grille* recouvre des conceptions très variées, allant d'une infrastructure "anonyme" fournissant de la puissance de calcul à la demande — de la même façon qu'EDF fournit de la puissance électrique à la demande — aux coopérations entre plusieurs instituts qui s'agrègent alors en *organisations virtuelles*, en passant par tout problème de grande taille résolu en utilisant un grand nombre de machines. Le mot *grille* ("*grid*" en anglais) s'est tellement banalisé et a été employé à tort et à travers, qu'il n'est pas aisé de savoir ce qu'est réellement une grille.

La première définition attestée, largement répandue, et qui a fait autorité pendant quelques années, est celle donnée par Ian Foster et Carl Kesselman, co-fondateurs du projet *Globus* TM [32], héritier de *I-Soft* et Nexus, en 1999 dans leur ouvrage de référence [88] :

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

Sentant que cette définition était trop vague et collait de moins en moins à la réalité, les mêmes auteurs publient deux ans plus tard une mise à jour sous la forme de leur vision de l'« anatomie » [91] d'une grille :

> « *coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations.* »

L'aspect "puissance à la demande" a disparu pour faire place à un *partage coordonné*, notion qui est appelée *Work as One* TM (*sic* !) par les auteurs de *Legion* [99]. Devant la jungle régnant pour l'attribution de l'appellation *grille*, Ian Foster précise dans une chronique intitulée "*What is the Grid?*" [84] sa vision en trois points :

- des ressources coordonnées sans contrôle central ;
- l'utilisation de protocoles et d'interfaces standard, ouverts, et généralistes (*ie.* pas uniquement dédiés aux grilles) ;
- un service "non-trivial", c'est-à-dire que le service offert par une grille est supérieur à la somme des services fournis individuellement par ses constituants.

La réponse [94] de Wolfgang Gentzsch, responsable "grille" chez *Sun Microsystems*, notamment éditeur de *Sun Grid Engine* [31], ne s'est pas faite attendre ; il s'indigne de l'exclusion des approches centralisées, des solutions propriétaires, et de l'ambiguïté manifeste du dernier point.

En l'absence de définition universellement acceptée, nous posons notre propre définition d'une grille :

Définition 3.1 : ***grille*** — Nous appelons une *grille* un ensemble de ressources (processeurs, mémoires, disques, réseaux, instruments de mesure, périphériques d'entrée/sortie, etc.) distribuées sur plusieurs sites, domaines d'administration ou institutions, mises en commun pour une utilisation coordonnée.

Cette définition est volontairement peu restrictive et englobe des cas qui sont souvent exclus des définitions.

3.1.2 Architecture des grilles de calcul

Dans cette section, nous décrivons l'architecture des grilles de calcul. Les grilles sont constituées d'une agrégation des ressources disponibles : stations de travail, grappes, supercalculateurs, reliés par des réseaux SAN, LAN, ou WAN, selon la portée. Les caractéristiques principales des grilles sont la diversité — hétérogénéité des machines et réseaux — et l'utilisation de ressources *parallèles* (grappes, supercalculateurs) reliées par des techniques du *réparti* (WAN).

Ces ressources agrégées gérées de façon coordonnée sont à même de satisfaire les besoin sans cesse croissants des applications — besoins en puissance de calcul, quantité de mémoire, capacité de disque. Le type d'application est souvent utilisé pour qualifier le type de grille :

Les grilles de calcul — Une *grille de calcul* est une grille destinée à des applications gourmandes en calcul, c'est-à-dire dont le goulot d'étranglement est situé au niveau du processeur.

Les grilles de stockage — Une *grille de stockage* est une grille destinée à des applications gourmandes en stockage sur disque.

Les grilles d'information — Une *grille d'information* est une grille destinée à partager la connaissance.

Nous parlons de *grille locale* ou *grille privée* dans les cas d'agrégation de ressources sur un même site. Nous distinguons un cas particulier de grilles de calcul basés sur des serveurs de calcul dédiés auxquels ont envoie des calculs en mode client-serveur ; il s'agit alors de *metacomputing*, comme mis en œuvre par exemple dans DIET [56].

Spécificités des grilles. Les spécificités des grilles sont de combiner à la fois des réseaux haute performances (SAN) et des réseaux longue distance (WAN), des aspect du parallélisme et des aspects des systèmes répartis.

Les problématiques spécifiques des WAN sont la sécurité, la rupture de connectivité, et la performance. La sécurité (authentification et chiffrement) est un aspect crucial lors de la traversée d'un WAN. Les applications critiques ne sont pas les seules à pouvoir en bénéficier : l'authentification est nécessaire pour toutes les applications pour ne pas compromettre la sécurité d'un site. La rupture de connectivité introduite par les pare-feux est un aspect lié à la sécurité. Il s'agit alors de chercher un moyen "autorisé" et contrôlé de franchir la frontière d'un site, pour communiquer sans compromettre sa sécurité. Enfin, du point de vue des performances, la meilleure façon d'utiliser un WAN n'est pas forcément directement TCP/IP. Il existe parfois d'autres méthodes plus performantes, par exemple en compressant [114] les données sur les réseaux lents, ou en utilisant plusieurs connexion physiques pour une seule connexion logique — technique dite des "flux parallèles", notamment utilisée dans *GridFTP* [39].

La problématique principale des SAN vient du fait qu'ils sont rarement basés sur le protocole IP omniprésent ailleurs sur les grilles. Ceci rend délicat l'adressage et l'utilisation de ces réseaux, en demandant l'utilisation de mécanismes spécifiques à chaque réseau.

Un exécutif taillé pour les grilles se doit d'être capable d'utiliser toutes les classes de réseaux présents — SAN, LAN et WAN —, en tenant compte pour chaque type de réseau de ses spécificités.

3.1.3 Infrastructures logicielles pour les grilles

Dans cette section, nous réalisons un tour d'horizon des infrastructures logicielles existantes pour gérer des grilles de calcul. Ce sont essentiellement Globus, Legion, et Unicore.

***Globus Toolkit*™.** Le *Globus Toolkit*™ [87] est un boîte à outils logicielle développée par l'*Alliance Globus* [32] composée du laboratoire national d'Argonne (Illinois), de l'USC/Information Sciences Institute (Californie), les deux membres fondateurs, auxquels se sont ajoutés en 2003 l'université d'Edinburgh et le centre suédois pour les calculateurs parallèles (PDC). Globus tire son origine de Nexus [89, 90, 85], plate-forme de communication décrite à la section 2.2.3. En 1996, Nexus a été déployé sur plusieurs sites couplés pour une démonstration inédite ; ce fut l'expérience dite "*I-Way*" [67], pour laquelle a été développée un ensemble d'outils appelé *I-Soft* [86]. Il en a dérivé une version *générique*, c'est-à-dire non liée à *I-Way*, qui a été distribuée sous le nom de Globus.

Globus est un ensemble modulaire d'outils relativement indépendants les uns des autres. Chaque outil peut être utilisé seul ou avec d'autres. Les outils sont organisés selon trois catégories : gestion de ressources, gestion de données, service d'information, auxquelles s'ajoute l'aspect transversal de la sécurité. L'expérience *I-Way* a montré [74] que le premier problème au déploiement d'une application sur une grille est la connaissance des ressources existantes et disponibles. La solution proposée dans Globus consiste en la mise en place d'un annuaire de ressources qui enregistre les machines participant à la grille. C'est le MDS (*Metacomputing Directory Service*) [63, 62] La gestion des ressources est assurée par GRAM (*Globus Resource Allocation Monitor*) qui lance des processus sur une grappe ou une machine parallèle, et DUROC (*Dynamically-Updated Request Online Coallocator*), co-allocateur qui réalise des allocations groupées vers plusieurs GRAM pour des applications qui nécessitent plusieurs machines parallèles. La gestion des données est constituée de GASS (*Global Access to Secondary Storage*) qui rend les données accessibles à distance *via* une API utilisable depuis les applications, et *GridFTP* [39], version spécialisée de FTP pour les grilles. Les communications sont assurées successivement, selon les versions, par Nexus, Globus I/O, puis Globus XIO [55]. L'exécutif proposé aux application est MPICH-G2 [81].

À partir de la version 3 (Juin 2003), Globus est basé sur l'architecture OGSA (*Open Grid Service Architecture*) [83]. Globus 3 est conforme aux recommandations OGSI [175]. Il s'agit d'une encapsulation des services Globus dans des *Grid Services*, variantes de *Web Services*, ce qui rend l'accès à distance plus homogène et obéissant à un standard.

De très nombreux projets à travers le monde utilisent Globus pour la recherche et en production. Plusieurs industriels, dont IBM, fournissent un support à Globus.

Legion. Legion [140, 99] est un projet de l'université de Virginie, démarré en 1994. À l'origine, Legion devait être la plate-forme d'exécution de programmes *Mentat* [98], extension parallèle de C++. Il est très vite apparu que l'approche pouvait être généralisée et Legion s'est rapidement détaché de *Mentat*. Notons que les travaux sur *Mentat* semblent arrêtés depuis 1996 ; Legion est toujours très actif, et commercialisé par *Avaki Corporation* (anciennement *Applied MetaComputing Inc.*) depuis octobre 1999.

Legion fournit des services de stockage persistant, de gestion de processus, de communication, de sécurité et de gestion de ressource, services qui sont habituellement fournis par le système d'exploitation, ainsi que des services spécifiques aux grilles, tels que l'accès à des données distantes, le support de l'hétérogénéité, et un support multi-langage (C, C++, FORTRAN, Java). Legion fournit ces services dans un environnement intégré, cohérent, à la différence des services indépendants les uns des autres — et il est vrai un peu hétéroclites — de Globus.

Legion propose une abstraction des différentes ressources en une unique "machine virtuelle" [99] qui recouvre la grille. Cette machine virtuelle est basée sur le principe d'objets partagés sécurisés, stockés dans un espace de nommage global [60, 140]. Toutes les ressources sont de tels objets : fichiers, utilisateurs, processus, machines, ordonnanceurs, connexions, etc. Legion unifie tous les espaces de nommage en un unique espace appelé *context space*, largement inspiré de l'arborescence Unix. Cet espace est accessible depuis toute machine participant à une grille Legion, fournissant ainsi une image globale unique de toute les ressources depuis n'importe quel point de la grille

3.1.3.1 Autres plates-formes

À côté de Globus et Legion, d'autres projets ont vu le jour. Ce sont par exemple :

Unicore — Unicore [35] est un projet anglo-allemand de plate-forme pour grille de calcul. Unicore offre un langage de description d'enchaînement et de dépendances des applications, ainsi qu'un outil graphique pour construire de telles descriptions et les envoyer à un serveur de calcul. Unicore propose ses propres serveurs, et peu également utiliser le GRAM de Globus.

Sun Grid Engine — *Grid Engine* [31] de *Sun Microsystems* est une plate-forme qui prend en charge le lancement d'applications et la gestion multi-utilisateurs sur une grappe ou une grille privée.

Condor — *Condor* [129] est une plate-forme développée à l'université du Wisconsin. *Condor* ordonnance l'exécution des applications soumises en traitement par lot, sur des ressources diverses éventuellement éparpillées à grande échelle.

3.1.3.2 Conclusion sur les plates-formes logicielles pour les grilles

Nous avons présenté les principales infrastructures logicielles utilisées pour gérer les grilles de calcul. Ce domaine est en plein essor et, preuve du début de sa maturité, mène à des solutions commercialement viables comme l'attestent la commercialisation de Legion par *Avaki* et le support de Globus fourni par IBM.

Il existe une grande diversité de plates-formes chacune avec ses spécificités — des services à la carte pour Globus, une vision globale uniforme pour Legion. Cependant, nous avons vu que ces solutions sont essentiellement destinées à assurer le déploiement des applications à large échelle, en gardant l'architecture classique des gestionnaires de *jobs* déjà en vigueur sur les machines parallèles. Ces plates-formes maintiennent une liste des machines, un suivi de leur état, et la possibilité de lancer des applications avec décision des ressources à utiliser et transfert des données et codes. La structure interne de l'application n'est pas prise en charge. Ces plates-formes ne fournissent aucun modèle de programmation et pour la plupart n'intègrent aucun exécutif spécifiquement adapté aux grilles.

La suite de ce chapitre présente plusieurs modèles de programmation utilisés pour programmer les grilles de calcul : programmer une grille comme un système parallèle, comme un système réparti, puis avec un modèle de programmation généralisé combinant des aspects du parallélisme et du réparti.

3.2 Voir une grille comme un calculateur parallèle

Dans cette section nous présentons les méthodes et infrastructures utilisées pour programmer une grille comme un calculateur parallèle.

Du parallélisme aux grilles. Au départ vues comme des extensions de calculateurs parallèles, les grilles de calcul sont de part leurs origines couramment programmées de la même façon que les calculateurs parallèles. Aussi bien Globus que Legion sont issus d'extensions d'exécutifs pour langages parallèles. Les premières extensions telles que PACX-MPI [49] (*PArallel Computer eXtension*) développé à l'université de Stuttgart, avaient pour but initial de relier une *Intel Paragon* et une *Cray T3E*. La vision présentée est un "super-supercalculateur". La même approche a été retenue pour l'expérimentation américaine *I-Way* [67], avec une version de MPI [82] capable de relier les onze sites impliqués. Cette approche est toujours d'actualité, le principal exécutif pour programmer les grilles de calcul étant MPICH-G2 [81], implémentation MPI basée sur Globus capable de relier plusieurs machines parallèles et plusieurs sites.

Cette approche montre une grille comme un calculateur parallèle dont les nœuds sont répartis sur plusieurs sites. La "grille" est donc seulement visible au niveau des ressources utilisées ; au niveau applicatif, il y a très peu de différences par rapport à un système parallèle. Beaucoup d'applications parallèles basées sur MPI peuvent ainsi voir leur ensemble de ressources accessibles élargi, simplement en utilisant MPICH-G2 à la place d'une implémentation MPI pour machine parallèle. Certaines adaptations sont toutefois souhaitables pour transformer une application parallèle pour supercalculateur en application parallèle pour une grille. Ces adaptations sont :

- gérer la topologie du réseau, pour favoriser la localité géographique et privilégier les liens rapides par rapport aux liens lents ;
- gérer l'hétérogénéité des performances des différents nœuds à l'aide de mécanismes d'équilibrage de charge, même pour des problèmes réguliers ;
- gérer la tolérance aux pannes pour les applications longues, une grille constituée de nombreux nœuds étant plus susceptible de rencontrer un erreur en cours d'exécution qu'une seule machine parallèle.

Seule une application qui communique peu se transpose aisément d'une machine parallèle à une grille. Une application qui communique beaucoup, avec un grain très fin souffre de la latence introduite par l'éloignement des nœuds.

Problématiques des SAN et WAN. La particularité de l'architecture des grilles est la grande variété des réseaux qui la composent. L'exécutif parallèle doit donc être en mesure de gérer cette diversité, des SAN jusqu'aux WAN. Par exemple, MPICH-G2 inclut des méthodes de communication spécifiques pour les grilles. Il est capable d'utiliser les SAN à l'aide d'une implémentation MPI "locale" native fournie par le constructeur. Il utilise les WAN à l'aide de méthodes spécifiques ; par exemple, il peut activer la compression, et utiliser les infrastructures de sécurité de Globus pour l'authentification.

Toutes ces méthodes de communication sont prises en compte directement dans MPICH-G2, de façon spécifique, en dehors de tout cadre générique. Nous n'avons pas connaissance de plate-forme générique de communication qui offre ce type de service à un niveau générique, utilisable par divers exécutifs.

Discussion. La vision d'une grille de calcul comme un calculateur parallèle est une approche qui fonctionne dans certains cas. Une plate-forme telle que Globus assure les réservations dans les divers systèmes de traitement par lots, transfère les données et les exécutables, et procède aux lancements synchronisés sur plusieurs sites. L'application, programmée par exemple avec MPI, a le point de vue d'une machine parallèle, l'implémentation MPI se chargeant de gérer l'hétérogénéité et choisissant le réseau adéquat pour réaliser les envois de messages. Il est ainsi possible de déployer les applications parallèles existantes sur un ensemble de ressources élargi. Cependant, n'importe quelle application parallèle ne convient pas : la latence introduite par l'usage d'un WAN au lieu d'un SAN pour certains liens pénalise lourdement les applications qui échangent de nombreux messages. En pratique, les applications parallèles demandent une certaine adaptation pour fonctionner de façon satisfaisante sur une grille.

Cette approche, bien que fonctionnant, semble toutefois très restrictive. Une première étape d'extension est de chercher à offrir l'accès aux réseaux selon une interface abstraite générique de type parallèle. Une telle plate-forme générique parallèle pour les grilles doit permettre de porter n'importe quel exécutif parallèle sur les grilles, et pas seulement MPI, sans avoir à redévelopper toutes les méthodes spécifiques aux grilles pour chaque exécutif.

De plus, il ne nous semble pas légitime de réserver les grilles de calcul aux seules applications parallèles. D'autres applications que les applications parallèles sont susceptibles de profiter des grilles. C'est pourquoi, même si la vision d'une grille comme un système parallèle est une approche qui fonctionne, il nous semble qu'elle ne représente qu'une partie des possibilités permises par les grilles.

3.3 Voir une grille comme un système réparti

Dans cette section nous présentons les méthodes et infrastructures utilisées pour programmer une grille comme un système réparti.

Des systèmes répartis aux grilles de calcul. Les similitudes entre le calcul réparti et les grilles de calcul sont nombreuses. Dans les deux cas, des applications constituées d'objets ou de composants distribués sur des machines hétérogènes sont reliées par un intergiciel pour réaliser une action coordonnée. Les solutions du réparti semblent donc déjà prêtes à un déploiement à un déploiement sur une grille. Pourtant, force est de constater que les infrastructures logicielles des systèmes répartis ont été conçues avant tout pour les LAN ; elles ne sont guère adaptées à un usage sur WAN, et encore moins aux SAN. Le passage d'un monde réparti local à une grille de calcul apporte donc de nouvelles problématiques, en particulier une grande variété de réseaux différents à gérer ; en effet, comme décrit en section 3.1.2, la spécificité des grilles du point de vue des réseaux est la diversité, allant des SAN aux WAN. Nous présentons le positionnement des exécutifs du réparti par rapport à ces problématiques dans les paragraphes suivants.

Problématiques des WAN. Bien que la plupart des exécutifs du réparti soient construits sur IP, ceci ne leur garantit pas une exploitation directe des WAN. Par rapport aux LAN dont les exécutifs répar-

tis ont l'habitude, les WAN présentent en plus des problématiques de sécurité, de connectivité et de performances.

La sécurité, aspect crucial des WAN, est généralement correctement gérée dans beaucoup d'exécutifs répartis, très souvent assurée par le standard cryptographique TLS [68] (*Transport Layer Security*), très similaire à SSL propriété de *Netscape Communications*. L'implémentation *openSSL* [26] est couramment utilisée. La gestion des ruptures de connectivité IP introduits par les pare-feux est inégale. La solution généralement retenue consiste en l'utilisation d'un *proxy* ("mandataire") qui contrôle les requêtes en les analysant. Les *proxies* les plus courants sont pour HTTP (donc utilisables par SOAP et les *Web Services*), mais il existe également des *proxies* IIOP destinés à CORBA et RMI Java. Enfin, peu d'optimisations spécifiques pour les performances sur WAN sont faites dans les exécutifs du réparti. Par exemple, ICE de *ZeroC* est l'un des rares exécutifs à proposer nativement la compression des données.

Problématique des SAN. L'autre caractéristique des grilles du point de vue des réseaux est la présence de réseaux SAN issus du monde du parallélisme. Pour voir les grilles comme des systèmes répartis, il est souhaitable que les exécutifs répartis tels que CORBA ou RMI Java soient capables d'exploiter ces réseaux haute performance.

Pour CORBA, dès ses débuts des travaux ont été réalisés pour améliorer ses performances et le rendre utilisable sur SAN. Cependant, ces implémentations sont essentiellement expérimentales. Par exemple, OmniORB des laboratoire AT&T avait, dans sa deuxième version (1998), été adapté [130, 149] aux réseaux ATM [80] et SCI [108]. Les performances obtenues sont toutefois relativement médiocres, avec par exemple un débit s'élevant à seulement 6 Mbit/s pour un type structuré sur le réseau ATM de débit nominal à 155 Mbit/s. Dans cette version 2, OmniORB n'était pas très optimisé et réalisait en particulier beaucoup de copies en mémoire. Les versions suivantes sont plus performantes, mais les versions destinées aux réseaux haut débit ont été abandonnées. D'autres travaux CORBA sur SAN plus marginaux existent. Un portage de MICO sur VIA [70] a été réalisé [69]. Cette mise en œuvre a un débit jusqu'à 12 % plus rapide qu'une utilisation du même matériel Gigabit Ethernet avec TCP. Une version modifiée [123] de MICO pour éviter les copies améliore spectaculairement ses performances sur Gigabit Ethernet, de 50 Mbit/s à plus de 500 Mbit/s ; l'implémentation n'est cependant pas complète et le code n'est pas distribué. *CrispOrb* [111] est développé par les laboratoires *Fujitsu*. Il est destiné spécifiquement aux réseaux VIA, et a été testé sur *SC-net*, une émulation logicielle de VIA sur le SAN *Fujitsu Synfinity-0*. Le gain en latence est de 20 % par rapport à une utilisation du même matériel par TCP/IP. Les résultats des travaux existants sont encourageants quant à la possibilité de réaliser une implémentation CORBA sur ce type de réseaux, mais semblent très incomplets. Chaque implémentation est spécifique à un type de réseau particulier ce qui restreint l'utilisation potentielle. De plus, ce sont des orientations marginales : le développement d'OmniORB sur les réseaux rapides a été arrêté après un prototype confidentiel, le prototype MICO/VIA n'a pas abouti, le prototype MICO zéro copie n'est ni complet ni distribué, et *CrispOrb* cible une machine en particulier. Aucun de ces travaux n'a réellement abouti à une implémentation CORBA complète utilisable sur SAN.

Il existe également des travaux destinés à rendre les RMI Java capables d'utiliser les SAN. L'implémentation Java originale ne sait en effet utiliser que TCP/IP et utilise une coûteuse sérialisation elle-même écrite en langage Java ; les latences couramment observés sur Ethernet-100 sont de l'ordre de 1000 à 2000 μs. Des travaux [132, 159] visant à porter les RMI Java sur *Fast Messages* [148] et Panda [161] et l'utilisation de code natif plutôt que du code binaire Java ont montré qu'il est possible d'abaisser cette latence à environ 35 à 50 μs en utilisant Myrinet.

Discussion sur les grilles en tant que systèmes répartis. Les travaux visant à porter les exécutifs du réparti sur les SAN semblent encourageants, certes, mais les résultats sont médiocres et constituent des orientations marginales des développements. Aucun de ces travaux n'a abouti à une généralisation de la fonctionnalité vers une version officielle ou dans la spécification. Aussi bien pour CORBA que pour RMI Java, l'utilisation des SAN est anecdotique et réservée à des versions modifiées d'exécutifs à la diffusion confidentielle. Cependant, même si ces orientations sont marginales dans les faits, la possibilité d'adapter les exécutifs du réparti tels que CORBA ou RMI Java aux réseaux SAN semble attestée. L'exploitation des WAN est mieux prise en compte. Cependant, la plupart des travaux restent

des initiatives isolées et n'utilisent pas de mécanismes généraux, à part l'omniprésent TLS/SSL pour la sécurité.

Nous observons que les solutions du réparti, bien adaptées aux LAN, sont moins à l'aise sur la diversité des réseaux des grilles. Pourtant, de nombreuses solutions initialement réservées au réparti (sécurité, notion de *proxy*, couplage faible entre les nœuds, etc.), et absentes du parallélisme, sont indispensables pour les grilles. Il est donc intéressant de transférer ces compétences vers des solutions logicielles spécifiquement conçues pour les grilles.

Comme pour la vision d'une grille comme un système parallèle, voir une grille comme un système réparti permet d'utiliser une classe d'applications existantes à une plus large échelle, sur des ressources plus diversifiées. Cette solution fonctionne dans certains cas, nécessite des adaptations des exécutifs dans d'autres cas. Toutefois, les modèles de programmation du réparti, s'ils conviennent effectivement à l'exploitation des ressources à large échelle, sont peu adaptés au calcul scientifique. En effet, beaucoup de codes de calcul scientifique sont parallèles. Il n'est pas envisageable d'exploiter efficacement une grille de calcul avec le seul modèle client/serveur du réparti en se privant de la programmation parallèle pour des applications générales de calcul scientifique.

3.4 Nouvelles possibilités des grilles

Cette section introduit des façons de programmer les grilles qui prennent en compte les spécificités des grilles et permettent d'aller plus loin que de les utiliser comme des systèmes simplement parallèles ou répartis. Nous introduisons dans cette section des méthodes qui considèrent les grilles comme une nouvelle classe d'architecture à part entière.

Au carrefour du parallélisme et des systèmes répartis. Nous avons présenté jusqu'ici deux façon de programmer les grilles : comme des systèmes parallèles ou comme des systèmes répartis ; chacune de ces deux approches fonctionne dans certains cas, et permet d'utiliser les grilles d'une façon restreinte. Cependant, chacune des deux visions n'est qu'un sous-ensemble des possibilités offertes par les grilles de calcul. Les grilles de calcul, en tant que plates-formes de calculs, rejoignent la problématique de la performance qui caractérise le parallélisme. Par l'utilisation de ressources hétérogènes, diversifiées, et surtout réparties géographiquement, les grilles rencontrent les problèmes habituels des systèmes répartis. À ces aspects s'ajoute la volonté d'ouverture et d'interopérabilité souvent mise en avant pour les grilles, notion peu répandue en parallélisme et habituelle en systèmes répartis. Il s'ensuit que les grilles ne peuvent pas se résumer à une vision qui ignore l'un ou l'autre des aspects. La performance des calculs est d'importance égale à la gestion de la répartition géographique.

Combiner parallélisme et réparti. Les deux points de vue du parallélisme et des systèmes répartis peuvent être pris en compte dès les modèles de programmation. L'aspect "réparti" des grilles est essentiellement son organisation à grande échelle. En effet, les grilles de calcul sont à même de servir de plate-forme de déploiement pour des applications qui n'étaient pas envisageables auparavant, de par leur taille, leur complexité et leur consommation de ressources. Ces nouvelles possibilités permises par les grilles s'accompagnent d'une complexité accrue des codes des applications. Une façon de gérer ces codes complexes est de les encapsuler dans des objets distribués de type CORBA ou RMI Java, ou des composants logiciels [173] tels que ceux de CORBA (CCM).

Cependant, les codes de calcul étant souvent parallèles, il est souhaitable de développer des modèles d'objets ou de de composants qui, en plus d'être répartis, prennent aussi en compte le parallélisme. Des exemples d'extension vers le parallélisme de modèles de composants ou d'objets sont PaCO [157], PaCO++ [152], GRIDCCM [153], CCA [43] (*Common Component Architecture*) ou encore *Data Parallel* CORBA [141].

Comme l'illustre la figure 3.1 pour GRIDCCM, l'intérêt est alors que chaque code est encapsulé dans un objet ou un composant, avec tous les avantages que cela procure. Par exemple, les codes peuvent être développés par des équipes différentes, dans des langages différents, avoir des évolutions et mises à jour indépendantes, et même des licences différentes. L'encapsulation du parallélisme fait

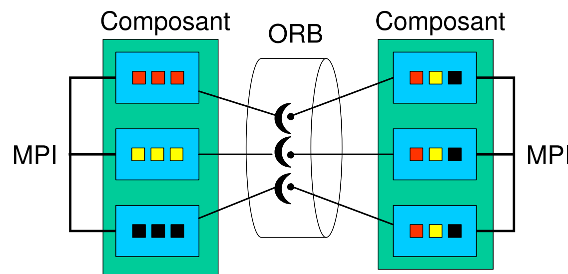

FIG. 3.1 – GRIDCCM : *des composants* CORBA *parallèles utilisent* CORBA *et* MPI.

que le choix d'un exécutif pour gérer le parallélisme est confiné dans le composant et n'est pas imposé à tous les autres composants ; l'utilisateur d'un composant à distance n'a pas à savoir quel est l'exécutif utilisé pour gérer le parallélisme.

Interactions. D'autres modèles combinent à la fois des aspects du parallélisme et du réparti. La combinaison des deux est d'autant plus tentante pour structurer les applications quand leur taille et leur complexité augmente. Les applications sont multiples, pour piloter et superviser un code à distance pour le contrôler en cours d'exécution afin d'agir sur les données ou les méthodes de calcul, pour visualiser les résultats en temps réel et interagir avec les calculs au travers de la visualisation, comme réalisé dans l'ACI GRID EPSN, pour récupérer le résultat d'un calcul pour le donner en entrée d'un autre code, comme fait couramment en couplage de codes comme par exemple PAWS [28]. À partir du moment où l'un au moins des codes est parallèle, nous sommes encore en présence d'une combinaison de parallélisme et de réparti. Il est envisageable de superviser un code à distance au travers de SOAP, ou encore de le rendre accessible à distance par *Web Services* ou CORBA.

Nouvelles problématiques. La caractéristique principale de ces modèles est de combiner à la fois des aspects du parallélisme et des aspects du réparti, aussi bien au niveau des ressources utilisées que du modèle de programmation. Nous pouvons les voir comme une généralisation et une rencontre des systèmes parallèles et répartis. La mise en œuvre de ces modèles combinant parallélisme et réparti amène de nouvelles problématiques :

- généralisation des ressources et méthodes de communication : il est souhaitable que tous les exécutifs — parallèles et répartis — soient capables d'utiliser toutes les ressources des grilles, telles que décrites en section 3.1.2. Aussi bien les exécutifs répartis que parallèles doivent être en mesure d'exploiter au mieux les réseaux de toutes les classes, des SAN jusqu'aux WAN. De plus, les méthodes de communications spécifiques pour certaines conditions doivent être utilisables indifféremment par tous les exécutifs.
- le besoin d'utiliser plusieurs exécutifs en même temps : les modèles de programmation qui mêlent réparti et parallélisme amènent le besoin d'utiliser plusieurs exécutifs en même temps dans la même application. Par exemple, un objet CORBA parallèle basé sur MPI nécessite l'usage de CORBA et MPI simultanément dans le même processus. De plus, ces combinaisons ne sont pas figées. Par exemple, il doit être possible d'utiliser CORBA et MPI dans certains cas, et CORBA et DSM dans d'autres cas. Il est souhaitable de pouvoir réaliser *a priori* n'importe quelle combinaison d'exécutifs.

Ces problématiques amenées par les nouvelles possibilités des grilles sont donc un sur-ensemble des problématiques de la vision d'une grille comme système parallèle et comme système réparti. Nous analysons ces deux problématiques dans le chapitre suivant.

3.5 Conclusion

Nous avons présenté dans ce chapitre le domaine des grilles de calcul. Notre vision des grilles est un assemblage de ressources parallèles et réparties hétérogènes, distribuées à large échelle. Nous avons présenté trois façons principales de programmer de telles architectures : utiliser les méthodes de programmation du parallélisme, utiliser les méthodes du calcul réparti, ou utiliser un modèle de programmation plus général composé d'un assemblage de méthodes du réparti et du parallélisme. Nous avons vu que les deux premières façons sont adaptées à la programmation des grilles dans certains cas particuliers, mais que chacune des deux approches est restrictive, couvrant uniquement une frange du spectre des possibilités des grilles. Des infrastructures logicielles pour gérer les communications dans ces deux cas existent mais semblent manquer de portabilité et de généricité : ce sont essentiellement des adaptations *ad hoc* d'exécutifs à des ressources particulières.

Les propositions récentes en matière de modèle de programmation pour les grilles mêlent à la fois des aspects du parallélisme et des aspects du calcul réparti. Cette approche englobant les deux premières, dans la suite de ce document nous nous intéresserons essentiellement à ce cas qui est le plus général ; chacun des autres cas — parallèle uniquement ou réparti uniquement peut être vu comme un sous-ensemble de ce qui est permis par le modèle général. Cette nouvelle façon de programmer les grilles apporte de nouveaux besoin en termes de communication et donc apporte de nouvelles problématiques. Les deux nouvelles problématiques principales soulevées par cette approche de la programmation des grilles de calcul sont :

- offrir des exécutifs selon plusieurs paradigmes, en utilisant éventuellement plusieurs exécutifs et/ou paradigmes en même temps ;
- rendre tous les exécutifs capables d'utiliser des ressources variées, elles-mêmes obéissant à divers paradigmes, le paradigme d'un réseau et d'un exécutif l'utilisant pouvant éventuellement être différents.

Nous nous intéressons plus particulièrement à ces problématiques dans le chapitre suivant.

Chapitre 4

Analyse des exécutifs communicants

Sommaire

4.1 Analyse de l'abstraction des ressources : portabilité et généricité 44
4.1.1 Une multitude de plates-formes génériques. 44
4.1.2 Limites de la portée de la généricité 44
4.1.3 Mondes fermés 46
4.1.4 Conclusion de l'abstraction des ressources de communication 47
4.2 Offrir plusieurs paradigmes simultanément 47
4.2.1 Plusieurs paradigmes sur un seul exécutif 47
4.2.2 Juxtaposer plusieurs exécutifs 48
4.2.3 Plates-formes de construction d'exécutifs 52
4.2.4 Plusieurs paradigmes simultanément : bilan 55
4.3 Conclusion 56

Dans ce chapitre, nous analysons les propriétés des exécutifs dans l'optique d'une utilisation par des applications sur des grilles de calcul. Nous nous focalisons essentiellement sur les *exécutifs communicants*, comme définis par la définition 2.10, et sur les applications basées sur des modèles qui mêlent des aspects du parallélisme et des aspects des systèmes répartis. Nous nous attacherons plus particulièrement à l'adéquation entre ces exécutifs communicants et les problématiques d'une programmation des grilles par combinaison de méthodes du parallélisme et des systèmes répartis, comme introduit au chapitre précédent.

Directions d'analyse. Nous menons notre analyse par rapport à deux caractéristiques essentielles, à savoir l'abstraction des ressources et la capacité à offrir plusieurs paradigmes :

Analyse *verticale* : abstraction des ressources — abstraire les ressources consiste en la séparation d'une interface et de son implémentation réelle. Elle confère aux exécutifs une meilleure indépendance vis-à-vis de matériels spécifiques. Dans l'empilement de couches logicielles, le niveau d'abstraction augmente de bas en haut ; nous appellerons donc cette analyse "analyse *verticale*". Elle fait l'objet de la section 4.1. Nous étudierons en particulier l'expressivité des interfaces abstraites et leur généricité.

Analyse *horizontale* : capacité à offrir plusieurs paradigmes — les modèles de programmation des grilles présentés au chapitre précédent n'imposent pas d'exécutif particulier mais au contraire laissent ce choix au programmeur de chaque partie d'application. Il en résulte une probable diversité d'exécutifs utilisés. De plus, les modèles d'objets et composants parallèles reposent intrinsèquement sur plusieurs exécutifs, l'un parallèle utilisé à l'intérieur d'un objet, l'autre réparti assure le couplage entre objets. Cette capacité à disposer de plusieurs exécutifs basés sur des paradigmes différents en même est analysée en section 4.2. Nous la nommerons "analyse *horizontale*".

4.1 Analyse de l'abstraction des ressources : portabilité et généricité

Notre première direction d'analyse des exécutifs communicants concerne l'abstraction des ressources, que nous appelons "analyse *verticale*". Nous avons présenté au chapitre 2 les mécanismes d'abstraction des ressources et les notions de *portabilité* et de *généricité*. Les plates-formes de communication génériques résolvent le problème de la quantité de code à écrire pour qu'un exécutif utilise divers réseaux, et de leur évolutivité vers de nouveaux types de réseaux. Nous pourrions penser qu'il s'agit d'une solution universelle pour disposer de tous les exécutifs sur tous les réseaux. Cependant, l'utilisation sur une grille les place dans des conditions qui sont inhabituelles. Cette section étudie les limites de l'approche générique pour la conception de plates-formes de communication : l'étendue de la généricité, le choix de l'interface générique, et l'interopérabilité entre les différents environnements.

4.1.1 Une multitude de plates-formes génériques.

Les plates-formes de communication génériques ont pour but de rendre automatiquement portable sur une multitude de réseaux tout exécutif qui l'utilise. Pourtant, la réalité est toute autre, et nous observons une profusion de plates-formes de communication qui se veulent toutes plus génériques les unes que les autres. Chaque plate-forme prend en compte un petit nombre de réseaux pour une utilisation spécifique. Par exemple, aussi bien Nexus que Globus I/O ciblent essentiellement MPI et TCP/IP, *Fast Messages* est portable uniquement sur TCP et Myrinet, Gamma se limite aux différentes variantes d'Ethernet.

Devant cette profusion de plates-formes génériques différentes, chacune apportant sa propre interface de programmation spécifique et son ensemble restreint de réseaux supportés, les auteurs d'exécutifs sont confrontés au choix crucial de la plate-forme à utiliser. Ce choix est d'autant plus crucial que les interfaces proposées par les plates-formes génériques sont spécifiques et introduisent inévitablement un lien fort entre la plate-forme et l'exécutif.

Pour permettre à un exécutif une portabilité sur un grand nombre de réseaux, il est finalement devenu courant de le porter sur plusieurs plates-formes génériques. Les exécutifs mettent alors en œuvre des mécanismes de portabilité pour gérer ces différentes plates-formes de communication, alors que ces plates-formes sont censées justement prendre en charge entièrement la portabilité. Chaque plate-forme contient alors des pilotes pour gérer quelques réseaux, et l'exécutif contient lui-même des pilotes pour gérer la portabilité sur les différentes plates-formes. L'ensemble s'en trouve finalement complexifié, et des problématiques de portabilité sont tout de même prises en compte dans les exécutifs malgré l'utilisation de plates-formes de communication génériques.

4.1.2 Limites de la portée de la généricité

Nous expliquons cette profusion de plates-formes toutes plus génériques les unes que les autres par une limite de la portée de la généricité, limite que nous analysons dans cette section. Même si un environnement générique est réutilisable dans plusieurs situations, il est plus adapté à certains cas qu'à d'autres. En effet, la motivation qui a conduit à chaque fois au développement d'une nouvelle plate-forme est que, à chaque fois, celles qui existaient ne convenaient pas par manque de généricité ou de facilité de portabilité.

Un pari sur l'interface. Même si l'objectif de généricité est louable, il est difficile à atteindre en pratique. Abstraire plusieurs interfaces de communication en une seule oblige à faire des choix. Ces choix cruciaux constituent un pari : passage de message, invocation à distance, mémoire virtuellement partagée, quel est le paradigme de communication le plus générique ? De plus, chaque paradigme peut être découpé en plusieurs variantes (ex.: messages synchrones ou asynchrones). Au final, il faut prendre certaines décisions qui peuvent sembler arbitraires dans certains cas. Nous analysons l'impact de ces choix sur la généricité, en nous basant sur trois exemples : Madeleine, Nexus et ADAPTIVE.

Le modèle Madeleine : RPC pour le parallélisme. Madeleine (décrite à la section 2.2.3) est initialement la bibliothèque de communication destinée à être utilisée par PM^2. Elle a depuis été utilisée par d'autres exécutifs que PM^2 ; la généricité était d'ailleurs une priorité lors de la conception des versions 2 [8] et 3 [47]. Madeleine a par exemple été utilisée comme cible d'une implémentation MPI [46]. Cependant, bien que Madeleine cible le calcul distribué en général, nous remarquons qu'elle présente de nombreuses caractéristiques du parallélisme et non du réparti. Par exemple, elle est basée sur le concept de topologie statique, construite à l'initialisation du programme lancé sur un ensemble déterminé de nœuds. L'adressage se fait ensuite par numérotation logique des nœuds, comme pour MPI. Même si, à l'instar de MPI-2, l'introduction de la dynamicité est possible dans Madeleine et fait l'objet de travaux en cours, l'ajout ou le retrait de nœud risque d'être considéré comme une reconfiguration de la topologie et pas comme une opération habituelle.

Les mécanismes d'échanges de messages du modèle Madeleine sont également orientés dans une direction précise : l'appel de procédure à distance. Rappelons que la raison d'être de Madeleine est l'intégration des *threads* et des communications. Or l'appel de procédure à distance (LRPC — *Lightweight Remote Procedure Call* — en l'occurrence) est mieux adapté qu'une réception explicite en environnement *multi-thread*. PM^2 était originellement basé sur PVM, et Madeleine est apparue suite à des résultats [53] qui ont montré que les mécanismes de passage de messages traditionnels (PVM ou MPI) ne sont pas adaptés à l'implémentation de mécanismes RPC haute performance où la taille des messages peut n'être connue qu'en cours de réception. Ainsi, il ressort que le modèle de communication Madeleine est fondamentalement orientée vers le paradigme de programmation parallèle à mémoire distribuée et le paradigme de communication RPC pour lesquels il est parfaitement adapté. En revanche, une programmation de type réparti (avec connexion/déconnexion fréquentes) sans notion de topologie globale ni de canal ne serait pas adaptée à Madeleine.

Le modèle Nexus : cible de compilateurs parallèles. Nexus, présenté dans ce document en section 2.2.3, est à l'origine un support exécutif pour le langage FORTRAN M [92] ; il a été rapidement étendu à d'autres langages comme Compositional C++ [59] et HPC++ [115]. Les origines de Nexus en tant que support exécutif pour langages parallèles transparaissent dans sa philosophie de conception et son API. Son organisation autour du concept de *pointeurs globaux* est un avantage décisif pour être la cible de compilateurs, car c'est une simple extension du concept de pointeur manipulé habituellement par les compilateurs. Cependant, ceci devient un handicap lorsqu'il s'agit d'une utilisation par un humain, comme c'est le cas quand un exécutif doit être porté sur Nexus comme par exemple les premières versions de MPICH-G.

En effet, le concept de *pointeur global* implique que pour disposer d'une connexion entre deux nœuds, il est nécessaire d'avoir au préalable un lien TCP/IP (établi par le lanceur), de créer *startpoints* et *endpoints* puis de faire migrer les *startpoints*. Cette construction fastidieuse de la topologie est à la charge du programmeur. L'intérêt de découper lien par lien la topologie réseau dans ce cas n'est pas flagrant car il masque la topologie sous-jacente et rend difficile, le cas échéant, l'utilisation de mécanismes de communications collectives qu'il serait intéressant de rendre accessible dans l'API, en plus du multicast, seule opération collective supportée. Ceci est d'autant plus marquant si l'on considère que Nexus cible essentiellement les réseaux haute performance (il est porté sur les interfaces MPI, MPL, Intel NX, etc.) qui comprennent la notion de collection de nœuds. D'une certaine façon, Nexus impose une API qui, du point de vue de l'utilisateur, présente des caractéristiques du réparti (lien par lien, dynamicité totale, gestion de l'hétérogénéité) alors que sa cible privilégiée est constituée de réseaux optimisés pour le parallélisme.

Nexus présente donc un modèle adapté en tant qu'exécutif pour compilateurs parallèles, mais ne parvient pas à être une plate-forme générique pour supporter d'autres exécutifs — chose pour laquelle il n'a pas été conçu. Son modèle est définitivement inadapté à jouer le rôle d'un environnement générique pour tous types d'exécutifs.

ADAPTIVE (ACE™) : intergiciel réparti à objets. ADAPTIVE, décrit dans ce document en section 2.3.3, est une plate-forme générique pour le réparti basée sur une abstraction C++ des communications en objets réutilisables pour faciliter les tâches courantes en programmation orientée objets

répartis. ADAPTIVE est ainsi un environnement sur mesure pour TAO, implémentation CORBA temps-réel. Depuis, ADAPTIVE a été utilisé dans d'autres projets d'environnements orientés objets répartis.

Il a été tenté d'élargir la portée d'ACE avec PACE, la partie basse d'ACE dotée pour l'occasion d'une API Posix. Cependant, PACE n'en est pas pour autant une plate-forme "passe-partout", aussi bien au niveau des réseaux supporté que de l'API présentée — uniquement une API du réparti —, et reste majoritairement utilisé pour des intergiciels à objets répartis.

Bilan : portée de la généricité. Comme nous venons de le voir sur les exemples de Madeleine, Nexus et ACE, la généricité d'un environnement de communication est une notion difficile à définir dans l'absolu en dehors de tout contexte. Aucun de ces environnements de communications n'est générique et portable dans tous les cas, c'est-à-dire supportant tous les réseaux et utilisable par n'importe quel exécutif communicant. Madeleine et ACE sont deux réussites car utilisés par plusieurs projets avec succès. Cependant, comme nous l'avons mis en évidence, leur généricité se cantonne pour chacun à un domaine particulier où ils excellent mais ne permet pas un franchissement de la frontière entre parallélisme et réparti. Madeleine apporte une bonne portabilité sur les principaux réseaux utilisés en parallélisme sur grappes en étant conçue dans l'esprit du parallélisme : mettre la haute performance à disposition des exécutifs construits au-dessus, grâce à une API qui permet d'exprimer les besoins. C'est ce qui explique le succès de Madeleine. ACE est un succès grâce à l'excellente gestion du principal problème en réparti : la portabilité ; ACE est en effet portable sur plusieurs dizaines d'architectures.

Quant à Nexus, c'est l'exemple typique d'un exécutif particulier auquel on a voulu faire jouer le rôle d'un environnement générique de communication, rôle plus générique que celui pour lequel il était prévu, d'où l'échec prévisible. L'adéquation avec l'utilisation typique est donc le facteur déterminant pour décider du succès d'un environnement générique de communication.

4.1.3 Mondes fermés

La portabilité et la généricité ont pour but de faciliter la réutilisation de logiciels, d'une architecture à l'autre et d'un exécutif à l'autre. Cependant, pour que l'utilisation d'environnements génériques et portables puisse se répandre, ils doivent satisfaire un troisième critère : l'interopérabilité. Dans ce contexte précis, l'interopérabilité consiste en la capacité d'un environnement de communication à communiquer avec des applications et/ou exécutifs qui ne sont pas basés sur cet environnement.

Les environnements pour le parallélisme à mémoire distribuée, tels que Madeleine [8] ou *Fast Messages* [148], sont conçus dans l'optique de codes SPMD fermés, c'est-à-dire que les nœuds communiquent uniquement entre eux, sans que les communications sortent du "monde" formé par les n nœuds. Ces environnements optimisent leur protocole de communication en supposant que les n nœuds utilisent le même environnement. Ils ne définissent aucun moyen d'interconnexion de plusieurs environnements. Du fait de l'adressage par numérotation des nœuds, il est même impossible de nommer une ressource extérieure. Cette supposition est cependant légitime dans la mesure où elle permet des optimisations particulières, ce qui est en accord avec les buts du parallélisme.

Les environnements de communication pour le réparti résolvent le problème de la portabilité par rapport aux système d'exploitation et à l'architecture de la machine, mais restent très majoritairement liés à IP, la langue commune de beaucoup de réseaux du monde réparti. Il y a très peu d'ouverture vers les réseaux du monde du parallélisme, comme les SAN, qui ne sont pas basé sur IP. En fait d'interopérabilité, il s'agit plutôt d'imposer un protocole commun. Enfin, il existe des environnements "avancés" du monde du réparti qui ne sont pas interopérables avec IP ; c'est le cas par exemple de x-kernel [106]. L'intégration avec des logiciels existants est problématique dans ce cas.

Nous pensons qu'un environnement de communication pour les grilles de calcul ne peut pas rester dans un "monde fermé" comme le font les environnements pour le parallélisme seul. À l'échelle d'une grille, les ressources sont variées et parfois imposées par des politiques locales des sites. Il est donc nécessaire de pouvoir dialoguer avec des applications sans être lié à un environnement particulier. L'abstraction des communications ne doit pas brider l'expressivité d'une façon qui interdise l'interopérabilité avec le monde extérieur. De plus, l'interopérabilité ne doit pas brider les performances sur SAN.

4.1.4 Conclusion de l'abstraction des ressources de communication

Nous avons étudié dans cette section l'abstraction des ressources de communication. L'abstraction — une interface *abstraite* pouvant être incarnée en plusieurs variantes — permet d'atteindre deux buts : portabilité et généricité.

Alors qu'il est relativement aisé de définir une abstraction pour la *portabilité* d'un seul exécutif, la définition d'une interface *générique* — *ie.* réutilisable d'un exécutif à l'autre — est délicate. Lors de sa conception, il est nécessaire de faire certains choix : privilégier la performance ou l'interopérabilité, choisir une orientation parallèle ou répartie, ou encore définir une API de transmission de données. Cependant, ce sont des choix qui relèvent de l'application, donc hérités par les exécutifs. Figer de tels choix dans une abstraction revient à donner une coloration particulière à l'abstraction — parallèle ou répartie, échange de messages ou RPC, datagrammes ou flux, etc. — ce qui constitue un pari sur l'utilisation future. Cependant, il semble difficile de s'affranchir de ces choix, et donc de cette limitation de la généricité. Toute interface abstraite générique a donc un champ d'action bien déterminé, principalement par son paradigme, et ne semble pas pouvoir être utilisée pour n'importe quel exécutif.

Il en résulte une grande diversité dans les plates-formes de communication, et il est parfois difficile de distinguer réellement la frontière entre environnements génériques et spécifiques ; en effet, puisqu'il n'existe pas de réel standard sur les interfaces abstraites de niveaux générique, chaque plateforme propose sa propre interface. C'est pourquoi il nous semble que, autant que possible, il est souhaitable de ne pas lier un exécutif à une plate-forme donnée si l'on veut pérenniser les développements.

4.2 Offrir plusieurs paradigmes simultanément

Après avoir étudié l'abstraction des ressources dans la première section de ce chapitre, notre deuxième direction d'analyse des exécutifs communicants concerne les différentes méthodes pour disposer de plusieurs paradigmes simultanément. C'est ce que nous appelons l'« analyse *horizontale* ».

Comme nous l'avons vu dans la section 3.4, les grilles de calcul, de part la puissance de calcul qu'elles sont capables d'offrir, ouvrent la voie à de nouvelles applications de plus en plus complexes. La conception de telles applications peut par exemple être basée sur de nouveaux modèles de programmation tels que les objets CORBA parallèles, les composants parallèles, ou d'autres modèles de couplage de codes parallèles. Le point commun entre ces modèles de programmation est l'utilisation d'une combinaison de mécanismes du paradigme réparti et de mécanismes du parallélisme (à mémoire partagée ou distribuée). Pour mettre en œuvre ces modèles de programmation, il est donc nécessaire de pouvoir disposer en même temps de plusieurs mécanismes de communication basés éventuellement sur plusieurs paradigmes différents.

Dans cette section, nous analysons la capacité des infrastructures logicielles existantes à offrir plusieurs paradigmes de communication et de programmation en même temps. Pour cela, nous explorons trois pistes : la possibilité d'utiliser plusieurs paradigmes sur un seul exécutif en "émulant" un paradigme sur un autre, l'utilisation simultanée de plusieurs exécutifs — un pour chaque paradigme — au sein d'une application, et les plates-formes de construction d'exécutif qui fournissent l'équivalent de plusieurs exécutifs en une seule entité logicielle.

4.2.1 Plusieurs paradigmes sur un seul exécutif

Devant la complexité des systèmes mis en œuvre dans les grilles, la tentation est grande de simplifier en choisissant une solution basée sur un seul exécutif commun pour toutes les tâches, même si ces tâches reposent sur des paradigmes différents d'un point de vue logique. Nous analysons dans cette section les approches qui émulent plusieurs paradigmes sur un seul exécutif, en ramenant tous les paradigmes à un seul par l'intermédiaire de couches d'adaptation.

Programmation répartie sur exécutif parallèle. La façon la plus populaire pour programmer les grilles est sans doute l'utilisation d'une implémentation MPI pour la grille, comme par exemple

MPICH-G2 [81]. En effet, MPICH-G2 est un moyen commode de déployer des codes parallèles sur un ensemble de ressources élargi, comme présenté à la section 3.2. Cependant, lorsque les codes réalisent des opérations supplémentaires basées sur un autre paradigme, il est tentant de se reposer sur la facilité d'accès aux diverses méthodes de communications *via* MPI. C'est le choix qui a été fait dans le coupleur de codes PAWS [28], où les mises à jour de données sont assurées par des invocations de méthodes à distance. Dans PAWS, l'adaptation pour fournir des invocation de méthode à distance au-dessus de MPI est apportée par la bibliothèque *Cheetah* [22].

Le problème qui se présente alors est la façon d'encapsuler des requêtes dans des messages MPI. Ce problème a été l'objet d'une étude [53] dont la conclusion est qu'en RPC, on ne connaît pas *a priori* la taille des messages à recevoir, alors qu'en MPI il faut savoir ce qu'on va recevoir ; les RPC sur MPI introduisent donc soit une négociation du format qui se traduit par un surcoût de latence, soit des copies de données et une pénalisation du débit. Dans tous les cas, les performances sont inférieures à ce qu'obtiendrait une implémentation qui s'affranchirait du format de message contraint de MPI. Cette conclusion est valable pour *Cheetah* et pour toutes les mises en œuvre de RPC (paramètre et tailles aléatoires) au-dessus de MPI (demande un format prévisible des messages). De plus, cette approche offre un RPC sur MPI, mais il ne s'agit pas réellement d'un RPC de type réparti. En effet, la dynamicité reste contrainte par MPI et l'interopérabilité ne semble pas pouvoir être assurée. Cette solution apparaît donc comme une solution *ad hoc* à un problème spécifique.

Programmation parallèle sur exécutif réparti. Il est envisageable d'utiliser comme seul exécutif CORBA ou RMI Java pour réaliser toutes les communications. L'émulation de passage de messages et de topologie statique au-dessus RPC et client/serveur est en effet plus facile que l'inverse. Cependant, cette approche présente un inconvénient majeur : elle impose les contraintes des exécutifs répartis à toutes les communications, comme décrit en section 3.3. Que l'exécutif choisi soit CORBA, RMI Java ou ICE, un unique exécutif réparti n'est pas en mesure d'exploiter les SAN. Il ne semble pas non plus satisfaisant d'adapter spécifiquement un tel exécutif aux SAN pour l'utiliser pour toutes les communications sur une grille. Cette approche ne nous semble de toute façon pas pertinente à cause de son surcoût logiciel, de ses probables faibles performances, et de son incapacité à offrir une vision collective des réseaux du parallélisme aux exécutifs du parallélisme.

Plusieurs paradigmes sur un seul exécutif : bilan L'utilisation de plusieurs paradigmes au-dessus d'un seul exécutif amène des problèmes de maintenabilité des codes, de performance et de capacité à utiliser divers types de ressources. Les exécutifs sont en effet destinés à mettre en œuvre un modèle de programmation d'un intergiciel. L'interface qu'ils proposent est donc spécialisée pour cette utilisation, et n'est de ce fait pas suffisamment malléable pour pouvoir implémenter facilement un paradigme différent au-dessus. L'interface offerte par un exécutif est trop spécialisée ne peut pas être transformée efficacement en un autre paradigme. Offrir plusieurs paradigmes sur un seul exécutif amène donc à privilégier un paradigme et à considérer les autres comme "secondaires".

Pour fournir une interface de programmation aux applications selon un paradigme fixé, il est donc préférable de réaliser l'adaptation de paradigme à un niveau inférieur et pas au-dessus d'un exécutif.

4.2.2 Juxtaposer plusieurs exécutifs

Nous explorons dans cette section une deuxième proposition pour disposer de plusieurs paradigmes de programmation et de communication dans le même programme, à savoir l'utilisation simultanée de plusieurs exécutifs en même temps.

Nous avons vu dans la section précédente qu'émuler plusieurs paradigmes sur un seul exécutif amène inévitablement à privilégier un paradigme plutôt qu'un autre. Le paradigme proposé par l'exécutif choisi est favorisé, car implémenté de façon "native" par l'exécutif, alors que l'autre est défavorisé soit au niveau des performances, soit au niveau des ressources supportées. Il n'y a pas de raison pour systématiquement donner la priorité à un paradigme au détriment des autres. Pour résoudre ces problèmes, une solution intuitive consiste à utiliser plusieurs exécutifs, chacun fournissant alors de façon native un paradigme. Par exemple, si une application demande l'utilisation simultanée

de CORBA et MPI, il semble naturel de *juxtaposer* une implémentation CORBA et une implémentation MPI.

L'utilisation simultanée de plusieurs exécutifs qui ne sont pas prévus pour cohabiter n'est cependant pas triviale. En effet, chaque exécutif est conçu indépendamment des autres en supposant être le seul exécutif dans le processus. Quand on utilise plusieurs exécutifs à l'intérieur d'un même processus, certaines suppositions faites par les concepteurs des exécutifs ne sont plus vraies. En particulier, la plupart des exécutifs considèrent qu'ils sont les seuls à accéder aux ressources et qu'ils peuvent en prendre le contrôle total. Quand plusieurs exécutifs partagent un processus, cette condition n'est plus remplie : il faut alors partager les ressources et y accéder coopérativement. La plupart ne sont pas conçus dans cette optique. Les conflits qui apparaissent sont essentiellement de trois types :

réentrance — certaines ressources ne sont pas prévues pour être utilisées en même temps par plusieurs codes clients. La tentative d'utilisation en simultanée représente une erreur.

compétition — quand l'accès simultané est possible, souvent la concurrence n'est pas contrôlée, pouvant mener à des iniquités ou des situations de famine, ou à une compétition pour s'approprier des ressources limitées.

choix global — certains paramètres choisis par l'exécutif sont imposés à tout le processus qui le contient. Deux exécutifs différents peuvent faire des choix différents et incompatibles, ce qui empêche leur cohabitation.

Les sections qui suivent présentent les conflits qui surgissent pour l'accès aux réseaux lors de la juxtaposition d'exécutifs, les conflits dans la gestion des processus, et enfin quelques solutions spécifiques pour des cas particuliers.

4.2.2.1 Conflits d'accès aux réseaux

Il est probable que des conflits surviennent si différents exécutifs juxtaposés utilisent les ressources comme s'ils en avaient l'accès exclusif. Les réseaux sont sans conteste la ressource la plus délicate à gérer dans la juxtaposition d'exécutifs communicants. Les conflits peuvent avoir lieu principalement pour l'accès aux réseaux haute performance, mais aussi pour l'accès aux réseaux locaux par TCP/IP. Les conflits se situent au niveau du choix des pilotes, de l'accès exclusif, de ressources limitée, ou de la gestion des signaux.

Certains réseaux sont utilisables par différents pilotes qui ne peuvent pas être utilisé en même temps ; le pilote est un choix global à la machine. Par exemple, Myrinet peut être utilisé par GM [136], *Myrinet Express* [137], ou BIP [156], mais un seul en même temps ; si par exemple un exécutif est construit sur GM et un autre sur BIP, il ne sera pas possible d'utiliser les deux exécutifs en même temps.

Il existe des méthodes d'accès au réseau qui sont exclusives, c'est-à-dire qu'elles ne sont pas réentrantes : à tout moment un seul client peut utiliser le réseau. Si différents exécutifs essaient d'utiliser ce même réseau en même temps sans s'être concertés (ce qui est le cas si on se contente de juxtaposer), alors chaque exécutif est un client du pilote réseau. Le premier exécutif à demander l'accès l'obtient, les suivants se voient l'accès refusé. Il est alors impossible d'utiliser plusieurs exécutifs standard sur de tels réseaux. C'est le cas par exemple de Myrinet utilisé avec le pilote BIP.

Certains réseaux ont des ressources limitées, et il est facile d'épuiser ces ressources si plusieurs exécutifs ouvrent des connexions séparées ; il s'agit alors d'une compétition pour s'accaparer les ressources. Par exemple sur le réseau SCI, le nombre de projections de segments disponibles (et donc de connexions logiques) est très limité ; de plus, si on l'utilise avec une méthode *double buffering* pour optimiser les transferts comme le fait Madeleine, la consommation de ressources est encore plus importante. Dans la pratique, au-delà d'une grappe de quelques nœuds, Madeleine consomme toutes les ressources SCI et il n'est donc pas possible pour un autre exécutif d'utiliser le même réseau. La limitation des ressources peut également se produire avec des *sockets* TCP/IP dans le cas où la connexion traverse un pare-feu. En effet, une solution souvent adoptée pour franchir un pare-feu est d'ouvrir un intervalle de ports sur le pare-feu pour laisser passer quelques connexions choisies et contrôlées ; typiquement, cet intervalle comprend un nombre de ports allant de la dizaine à la centaine. Si chaque exécutif ouvre des connexions dans cet intervalle, il est possible de remplir l'intervalle très rapidement.

Le nombre de connexions pouvant franchir un pare-feu est encore plus réduit quand elles doivent emprunter un tunnel ssh

Enfin, certains réseaux proposent un mode de réception par interruption. Les interruptions étant généralement gérées globalement pour le processus, si plusieurs exécutifs utilisent le réseau dans le même processus, il est possible qu'un exécutif reçoive les interruptions destinées à un autre exécutif. Dans le cas des *sockets* en particulier, les interruptions sont logicielles et prennent la forme des signaux Unix SIGIO, SIGPOLL et SIGURG envoyés au processus (ou éventuellement individuellement au *thread* dans le cas où des masques sont mis en place). Cette fonctionnalité est utilisée par exemple par le pilote ch_p4 de MPICH, implémentation MPI pour TCP/IP fournie en standard avec de nombreux systèmes, largement répandue et utilisée. Nous avons étudié l'exemple de la juxtaposition des exécutifs MPICH/ch_p4 et MICO dans le même processus. Nous avons observé que les communications de MICO déclenchent des signaux qui sont reçus par MPICH/ch_p4 ; ch_p4 traite alors le signal comme s'il lui était destiné, ce qui mène au final à un crash du processus.

Ces conflits d'accès au réseau qui surgissent à divers niveaux n'empêchent pas systématiquement la cohabitation d'exécutifs dans un processus. En effet, il existe des combinaisons qui fonctionnent. Cependant, les cas de conflits sont variés et touchent presque tous les types de réseaux, de Myrinet et SCI aux réseaux IP. Il semble déraisonnable de compter sur la chance et d'espérer tomber sur une combinaison d'exécutifs qui fonctionne ensemble. Il est nécessaire de prendre en compte ces conflits potentiels.

4.2.2.2 Conflits dans la gestion des processus

La juxtaposition d'exécutifs communicants sans intégration risque de causer des conflits dans l'accès à toutes les ressources dont les exécutifs prennent possession comme s'ils en étaient les seuls maîtres. C'est le cas particulièrement pour les ressources réseaux. Cependant, il y a également risque de conflit d'accès, essentiellement pour le *multi-threading* et la gestion des processus.

Il est maintenant de plus en plus courant pour les exécutifs d'utiliser le *multi-threading* dans leur implémentation. Ceci permet par exemple aux implémentations MPI de gérer les communications en tâche de fond sans avoir besoin d'appels MPI_Test pour les faire progresser, ou aux implémentations CORBA de gérer plusieurs requêtes entrantes simultanément. La contrepartie est que l'utilisation d'un exécutif introduit une dépendance vis-à-vis d'une bibliothèque de *multi-threading* particulière. Il est en effet délicat d'utiliser deux bibliothèques de *multi-threading* différentes en même temps au sein d'un même processus. Pour que l'application puisse elle-même utiliser le *multi-threading*, deux solutions existent : soit l'application *sait* quelle est la bibliothèque de *multi-threading* et utilise la même, soit l'exécutif fournit une abstraction des *threads* (par exemple omnithread pour OmniORB, MICOMT::Thread pour MICO, JTC pour ORBacus) qui assure la portabilité. Ces solutions peuvent convenir pour une application utilisant un exécutif, application et exécutif utilisant le *multi-threading*. Cependant, ces solutions ne semblent pas satisfaisantes pour une application qui utilise deux exécutifs, chaque exécutif pouvant décider d'utiliser une bibliothèque de *multi-threading* différente. Une approche classique consiste à se limiter à l'utilisation des pthread définis dans la norme Posix 1003.1c et relativement répandus dans les systèmes d'exploitation actuels. Cette approche ne résout cependant pas les cas où l'exécutif souhaite explicitement utiliser un certain type de *threads* systèmes (cthreads sur Mach, sprocs sur IRIX et Solaris, LinuxThreads, NGPT ou NPTL sur Linux) ou des *threads* de niveau utilisateur tels que Marcel [53] ou Green Threads.

En supposant que la bibliothèque de *threads* soit choisie, la façon de gérer les *threads* en environnement multi-exécutif est également problématique. Si l'on juxtapose deux exécutifs, l'un se basant sur une attente active (tel que Kaffe), l'autre se basant sur une attente bloquante (tel qu'OmniORB), alors l'inégalité est flagrante : l'exécutif qui effectue l'attente active mobilise la grande majorité du temps du processeur. Si la bibliothèque de *multi-threading* fournit en plus des mécanismes de priorités, alors en combinant les variantes d'attente active ou passive, et les niveaux de priorités, il est possible qu'un exécutif tombe en famine, *ie.* qu'il ne soit jamais ordonnancé.

Les exécutifs qui implémentent un modèle de programmation proposent souvent un modèle de déploiement associé. Par exemple des applications qui utilisent un exécutif implémentant la norme

MPI version 1 doivent être lancées par la commande `mpirun`, MPI 2 et PVM comprennent la notion de *spawn* (lancement dynamique de nouveaux processus *par* MPI *lui-même*), les composants CCM sont chargés et lancés par le démon *ServerActivator* de CORBA, ou encore les exécutifs HLA sont libres de gérer le déploiement comme ils l'entendent. Nous constatons donc qu'il est courant que l'exécutif veuille lui-même prendre en charge le lancement des processus. Dans le cas de l'utilisation simultanée de plusieurs exécutifs pour la même application, nous obtenons un conflit d'autorité sur le lancement des processus. Si aucun exécutif n'est modifié, le déploiement n'est tout simplement pas possible.

Outre le *multi-threading* et le déploiement, d'autres point sont à prendre en compte pour la juxtaposition d'exécutifs, comme la gestion de processus et l'accès à la mémoire. Par exemple, si un exécutif décide d'effectuer un `fork/exec` (comme le fait *Certi* [147]), que se passe-t-il pour les autres exécutifs dans le même processus ? Si une bibliothèque dépend d'un allocateur de mémoire particulier (comme BIP), comment être sûr que tous les codes l'utilisent ?

Nous constatons que, même si une grande partie des conflits concerne les accès aux réseaux, les aspects transversaux comme le *multi-threading* ou le lancement des processus amènent aussi des points litigieux. En pratique, certaines combinaisons fonctionnent, mais il est très difficile de prédire à l'avance si deux exécutifs pourront cohabiter. De plus, quand la cohabitation est possible, les performances peuvent être médiocres pour l'un ou l'autre ou les deux exécutifs. Les cas de conflits sont nombreux ; la variété des ressources et l'utilisation d'un grand nombre d'exécutifs différents multiplie par autant les probabilités de tomber sur un cas litigieux.

4.2.2.3 Intégrations spécifiques

Devant ces problèmes qui surgissent lors de la juxtaposition d'exécutifs, la solution employée dans certains cas consiste en la modification locale des exécutifs voulus pour les faire cohabiter. Il s'agit alors d'une intégration spécifique, qui résout les problèmes au cas par cas, ou d'une détermination de façon empirique d'une combinaison qui fonctionne.

Cette solution a été retenue pour PaCO [157] qui a besoin d'une implémentation CORBA et MPI. La combinaison retenue est composée de MPICH [100] et MICO [25]. Ni l'un ni l'autre ni PaCO n'utilise le *multi-threading*, de sorte que MPICH et MICO n'accèdent pas au réseau en même temps mais à tour de rôle. De plus, seul les réseaux TCP/IP sont exploités. Cette solution repose sur des versions particulières d'exécutifs, ce qui limite la maintenabilité et l'extensibilité.

Le successeur de PaCO, PaCO++ [152], est conçu sur une base portable, avec des pilotes pour choisir entre différentes implémentations CORBA, plates-formes de communication pour le parallélisme, et bibliothèques de *multi-threading*. C'est à l'utilisateur que revient la tâche de trouver une combinaison CORBA +MPI *threads* qui fonctionne. En pratique, la combinaison utilisée est OmniORB [45], MPICH-GM, et le *multi-threading* de PaCO++ est basé sur `omnithreads`, l'interface de portabilité pour le *multi-threading* d'OmniORB. Par chance, la version de MPICH sur GM n'utilise pas de *threads*. Pour les réseaux, OmniORB utilise exclusivement TCP/IP et MPICH-GM exclusivement Myrinet, ce qui limite les conflits. Au final, les combinaisons retenues en pratique pour PaCO++ utilisent CORBA et MPI sur deux réseaux différents — réservant le réseau rapide à MPI, et simplement TCP/IP pour CORBA —, et évitent les versions de MPI qui utilisent le *multi-threading*, comme par exemple MPICH/*Madeleine*.

Même dans les cas où la juxtaposition est possible, la flexibilité et la portabilité sont médiocres : pour supporter un nouveau type de réseau, il est nécessaire d'adapter tous les exécutifs — au passage en dupliquant beaucoup d'efforts — et de, à nouveau, étudier leurs interactions sur la nouvelle configuration. Il en est de même pour des méthodes de communication avancées spécifiques aux grilles telles que la compression ou l'authentification : les différents exécutifs doivent être adaptés un par un. La solution d'une intégration spécifique est donc adaptée à une expérimentation pour faire la preuve d'un concept, mais n'est pas satisfaisante pour une exploitation plus poussée.

4.2.2.4 Conclusion sur la juxtaposition d'exécutifs

Bien qu'intuitive pour disposer de plusieurs paradigmes de programmation et de communication simultanément, la méthode qui consiste à juxtaposer plusieurs exécutifs dans le même processus

souffre de nombreux défauts. Tout d'abord le risque de conflit d'accès aux ressources n'est pas négligeable, surtout en ce qui concerne les réseaux haute performance. Ensuite, même si par chance la cohabitation fonctionne, il est probable que les performances des communications soient sub-optimales. Enfin, même dans le cas d'intégrations spécifiques qui règlent ces problèmes au cas par cas, la portabilité est restreinte, demande une duplication d'efforts de portabilité, et ces intégrations sont peu évolutives vers de nouvelles méthodes de communication. Régler de cette façon les problèmes de cohabitation entre exécutifs pour un plus grand nombre d'exécutifs sur un plus grand nombre de plates-formes amène un trop grand nombre de combinaisons à examiner pour que cette solution soit considérée comme raisonnable.

4.2.3 Plates-formes de construction d'exécutifs

Cette section présente une troisième solution au problème d'offrir à une application plusieurs paradigmes de communication et de programmation simultanément. Il s'agit de l'utilisation de plates-formes de construction d'exécutifs, dont certaines prennent en compte dès leur conception la possibilité d'offrir plusieurs protocoles et/ou exécutifs simultanément. Ces plates-formes ont pour but d'être génériques et proposent donc la plupart du temps plusieurs paradigmes de communication et de programmation. Ces plates-formes sont basées sur une construction par *composants* (définition 2.13, page 15). Les composants permettant intrinsèquement l'instanciation multiple, ces plates-formes sont de bons candidats pour résoudre le problème de l'utilisation simultanée de plusieurs paradigmes par une même application.

4.2.3.1 Plates-formes à composabilité arbitraire

Des environnements tels que x-kernel [106] ou DAT [73] ont fait le pari de reconstruire entièrement des protocoles à l'aide de composants. Ces environnements partagent la vision d'une construction *à la carte* des protocoles en partant d'une bibliothèque de briques de base. L'assemblage est orchestré par l'environnement, en suivant les instructions données par l'utilisateur. La force de ce modèle est sans conteste que sa flexibilité autorise la construction de protocoles quelconques qui peuvent cohabiter, chaque composant pouvant servir dans plusieurs assemblages en même temps. Fournir en même temps par exemple un paradigme de passage de messages et d'appel de fonction à distance est naturel pour ce modèle.

Performances. Cependant, cette méthode de décompositions en petites entités amène le délicat problème de la performance. En effet, transmettre les messages d'un composant à l'autre ajoute inévitablement un surcoût, et diminuer le grain de décomposition pour gagner en flexibilité et réutilisabilité augmente dans le même temps le surcoût introduit. Les auteurs de x-kernel prétendent que le surcoût est très faible en comparaison du service ajouté par la décomposition. x-kernel propose de repartir d'une page blanche pour s'affranchir des contraintes de l'existant, la pile IP fournie par l'OS en particulier, et obtient d'excellentes performances. Il faut noter que ces performances de x-kernel datent de 1992 et que les piles IP des OS ont fait d'énormes progrès depuis lors, réduisant d'autant l'avantage de x-kernel. Depuis, x-kernel existe d'ailleurs en version également utilisable en espace utilisateur, avec des performances bien plus médiocres dans ce cas. Ceci conforte l'idée que le gain qu'apportait x-kernel en performances était plutôt lié au fait qu'il proposait une alternative aux piles IP médiocres de l'époque, et non à son approche par micro-protocoles. De plus, certains aspects problématiques des performances sont éludés car les performances n'ont été étudiés que dans les cas favorables : le point fort de x-kernel est qu'il permet une "remontée" des messages sans changement de contexte à la réception ; or, justement, les chiffres publiés se limitent à des protocoles de type RPC, cas favorable quand l'environnement lui-même est basé sur le RPC.

L'utilisation de composants assemblables par une tierce partie apporte inévitablement un léger surcoût dû aux indirections lors de communications inter-composants. L'influence de ce surcoût dépend essentiellement du *grain* de décomposition, c'est-à-dire la taille des composants, qui détermine la proportion de temps passé dans le code utilisateur et dans le code de gestion de l'assemblage des

composants. En réalité, la décomposition en composants à grain fin apporte un surcoût perceptible. Les travaux anciens [61] comme les travaux récents [73] mettent en évidence un surcoût, qui est chiffré actuellement de 5 à 30 % rien que pour la traversée de l'environnement lui-même dans le cas d'un protocole de type TCP/IP. Ce surcoût peut cependant être tolérable en contrepartie des fonctionnalités gagnées.

Problème de la définition de l'interface. S'il ne fait pas de doute que l'approche par composants est un atout pour la portabilité, elle n'est pourtant pas la panacée pour la flexibilité. Chaque plate-forme de composant apporte sa propre vision. Ainsi, DAT impose par son interface virtuelle de protocole que tous les protocoles soient basés sur des messages délimités avec API asynchrone ; il n'est pas prévu qu'un composant soit basé sur un autre modèle, des flux continus par exemple, ou une réception par messages actifs ou RPC. Ce type de limitation est encore plus flagrant dans x-kernel : puisque le principe de composabilité "tout-sur-tout" est la base de sa conception, absolument tous les micro-protocoles implémentent et utilisent une interface imposée unique.

La possibilité de composer *arbitrairement* les composants amène donc une limitation drastique : l'interface est *figée*, c'est-à-dire que tous les composants utilisent la même interface. Même s'il est possible de donner une sémantique différente (fiabilité, routage, etc.) à une même interface, les variations ne sont guère diversifiées. L'expressivité n'est donc pas très grande : ces environnements sont génériques dans la mesure où l'on se limite au modèle prévu, et même s'il est possible théoriquement de réaliser de nouveaux composants à l'infini, ces nouveaux composants doivent exprimer l'interface imposée pour être utilisable dans la plate-forme. Il en résulte que, même si ces environnements permettent une composition entièrement libre des composants, ils sont adaptés à la réalisation de protocoles de relativement bas-niveau, mais n'ont pas une expressivité suffisamment riche pour réaliser des exécutifs de plus haut niveau.

Interopérabilité. Enfin, on remarquera que l'interopérabilité de tels environnements à composabilité arbitraire est problématique. Quand un message est transmis, le récepteur ne sait pas quel assemblage a été utilisé en émission ; par conséquent, la description de l'assemblage est transmise avec le message. Il en résulte que par exemple, la version d'IP reconstruite dans x-kernel n'est pas conforme à la norme — et ne *peut pas* être conforme, du fait des informations supplémentaires à transporter — donc n'est pas interopérable avec les réseaux IP classiques. DAT souffre du même problème d'interopérabilité en ne fournissant que des protocoles qui rentrent dans le moule qu'il impose (TCP/IP n'en fait pas partie). Nous pouvons généraliser ce manque d'interopérabilité à tout environnement à composants qui envoie des messages auto-décrits, *ie.* chaque message contient une description de l'assemblage des composants qui l'a construit ; les environnements qui se reposent sur des assemblages connus des deux parties ne présentent pas ce défaut.

En conclusion, de tels environnements à composants où la composabilité est arbitraire semblent parfaitement adaptés à la conception et l'expérimentation de prototypes de protocoles réseau. Cependant, ils semblent peu adaptés, sous cette forme, à une utilisation à grande échelle, par manque d'interopérabilité avec l'existant et manque de généricité. Enfin, nous noterons que l'interface fixée des composants interdit la réalisation d'interfaces de haut niveau, et donc d'exécutifs complets. Il est possible de réaliser diverses variantes de RPC bas-niveau avec x-kernel, mais une implémentation CORBA conforme à la norme viendrait se greffer par-dessus.

4.2.3.2 Environnements adaptables

Approche par réflexivité : l'exemple *Quarterware*. Dans l'approche précédente basée sur la composabilité arbitraire, l'encapsulation à tout prix rend les optimisations difficiles ; les interfaces figées contraignent inutilement les implémentations et peuvent même cacher des fonctionnalités que l'on aimerait voir exposées. Partant de ces constats, Ashish Singhai a proposé [169] une approche qui utilise la réflexivité, à savoir la capacité de découvrir l'interface proposée par un composant et d'observer le contenu des assemblages (composites) qui jouent le rôle de composants. Les composants n'ont alors

des interfaces ni uniformes ni figées et l'encapsulation n'est pas stricte. De cette façon, l'assemblage est réalisé à la volée, l'exécution peut dépendre de l'état pour des besoins d'optimisation, et on ne s'interdit pas d'observer le contenu de "procédures" pour en comprendre finement le fonctionnement.

Ces concepts sont implémentés dans le logiciel *Quarterware* [169], plate-forme malléable de construction d'exécutifs. Dans *Quarterware*, la réflexivité permet l'adaptation et la reconfiguration dynamique. En réalité, pour que le temps de calcul ne soit pas gaspillé à sans cesse calculer les assemblages, *Quarterware* propose un modèle d'exécutif à objets distribués, inspiré de CORBA et RMI Java, et est bâti sur ADAPTIVE [164]. Ce modèle général peut ensuite être spécialisé pour présenter une autre interface. Deux spécialisations ont été implémentées : un sous-ensemble de MPI, et un sous-ensemble de RMI Java. Les performances obtenues, en particulier la spécialisation MPI, sont bonnes sur TCP/IP : légèrement meilleures que l'antédiluvien MPICH/`ch_p4`, et équivalente à LAM. Les performances publiées sur ATM sont très moyennes. Du fait de l'utilisation d'ADAPTIVE, *Quarterware* est limité aux réseaux du réparti.

Discussion : granularité de décomposition. Pour améliorer la performance de systèmes à composants, il se pose le délicat problème du grain de décomposition. *Quarterware* fait le pari d'une décomposition fine, ce qui mène à un grand nombre de types de composants différents. Si on ajoute que dans ce modèle l'encapsulation n'est pas complète pour autoriser les optimisations, l'ensemble est complexe à maintenir et à faire évoluer. La complexité n'est pas dans les composants eux-mêmes mais dans la façon de les assembler. Elle est alors gérée explicitement par le concepteur d'exécutif. C'est d'ailleurs pour cette raison que les implémentations MPI et RMI Java de *Quarterware* ne sont pas complètes. La flexibilité et l'extensibilité apportées par cette approche semblent donc bien artificielles : ajouter un exécutif à *Quarterware* demande une masse de travail considérable, car il faut ré-implémenter l'exécutif complet à l'aide des composants *Quarterware*. La décomposition à grains fins avec la réflexivité relègue une grande partie de la complexité dans l'assemblage, ce qui ne fait que différer le traitement du problème.

La granularité choisie par *Harness* [134] (décrit en section 2.2.3) semble plus réaliste : l'expertise nécessaire pour développer un exécutif est encapsulée à l'intérieur d'un composant, et les concepteurs d'exécutifs sont libres d'employer la technique qui leur semble appropriée à l'intérieur de l'exécutif. Cependant, la démarche *Harness* est surtout ciblée sur l'adaptabilité et la dynamicité, laissant un peu de côté la diversité d'exécutifs et la performance. À proprement parler, *Harness* ne permet pas de faire cohabiter plusieurs paradigmes de communications puisqu'il offre seulement PVM et MPI qui sont tous les deux des exécutifs parallèles à mémoire distribuée et à passage de messages.

Nous pensons que l'approche introduite par *Harness*, à savoir une plate-forme dans laquelle l'utilisateur branche sous forme de *plugin* des exécutifs et des codes applicatifs, est prometteuse pour fournir plusieurs exécutifs simultanément et ne demande qu'à être étendue vers le calcul réparti. À l'inverse, nous pensons qu'une approche du type de *Quarterware* basée sur une reconstruction de chaque exécutif n'est pas facilement évolutive.

4.2.3.3 Synthèse et conclusion sur la construction d'exécutifs

Les différents exécutifs ayant beaucoup de notions en commun, il semble judicieux de factoriser ces parties communes pour éviter de refaire des choses déjà faites. Ceci est d'autant plus vrai lorsque l'on désire utiliser plusieurs exécutif simultanément. Les diverses méthodes permettant de mettre en commun certaines fonctionnalités sont majoritairement basées sur le concept de composant logiciel.

Certaines approches (x-kernel, DAT) prônent des composants à interface uniforme pour une composabilité maximum. Comme nous l'avons vu, cette méthode est limitée à la réalisation de protocoles de bas niveau, l'interface imposée manquant cruellement d'expressivité pour réaliser des exécutifs complets et conformes aux standards. D'autres approches autorisent des interfaces différenciées pour les composants, menant à des assemblages complexes (*Quarterware*) ou simplement la mise en commun de l'accès aux ressources (*Harness*). Notons toutefois que pour ces deux dernières approches, les implémentations se sont focalisées sur les parties hautes délaissant les parties basses — *Quarterware* est conçu pour les réseaux de type réparti avec des performances sur ATM calamiteuses, *Harness* tourne

sur TCP/IP avec un portage sur VIA anecdotique — ce qui rend leur évaluation sujette à caution, car elles n'ont pas été mises en œuvre dans le cadre haute performance des SAN

De plus, le point commun à beaucoup de ces approches par composants est qu'ils nécessitent des re-développements lourds d'exécutifs, alors que ces exécutifs existent déjà par ailleurs. Nous pouvons légitimement nous poser la question de savoir s'il est "rentable" de demander de refaire des travaux déjà faits simplement pour ajouter la cohabitation. Le gain apparaît minime, au vu de ce qu'il est possible d'obtenir à moindre coût avec des intégrations spécifiques.

En conclusion, ces approches sont flexibles à première vue, mais ciblent soit exclusivement le bas niveau, soit se concentrent sur l'assemblage à haut niveau en demandant une duplication d'efforts de développement d'exécutifs. Aucune ne considère le problème dans son intégralité, du pilote réseau jusqu'à l'interface applicative. Certaines pistes sont toutefois très prometteuses, notamment le principe d'une composition à nombre de couches *a priori* inconnu apporté par x-kernel ou la notion de plate-forme où l'on branche les exécutifs en tant que *plugin* apportée par *Harness*. Cependant, aucune ne résout le problème d'un accès collaboratif aux ressources réseau parallèles et répartie ; à chaque fois, le problème traité est le cas de plusieurs exécutifs sur *un* réseau. De plus, presque toutes ces approches, bien que fournissant l'équivalent de plusieurs exécutifs en même temps, se cantonnent à un seul paradigme pour les exécutifs. La seule approche à réellement proposer des exécutifs selon plusieurs paradigmes dans un environnement flexible (*Quarterware*, qui combine RMI Java et MPI) a dû être simplifiée à l'extrême (deux exécutifs, en version partielle) et dénaturée (finalement, tout dérive d'un modèle commun d'objets répartis) pour être implémentable, sous une forme à l'évolution et la maintenabilité délicate. Même si les approches à composants semblent intéressantes pour construire des plates-formes de communication, reconstruire tous les exécutifs en fonction d'un modèle de composant fourni par une plate-forme particulière semble une approche délicate et complexe à maîtriser.

4.2.4 Plusieurs paradigmes simultanément : bilan

Nous avons étudié la capacité des infrastructures logicielles existantes à offrir plusieurs paradigmes en même temps à la même application. Nous avons considéré trois solutions : l'émulation de plusieurs paradigmes sur un seul exécutif, la juxtaposition de plusieurs exécutifs, et les plates-formes de construction d'exécutifs à composants. Si la juxtaposition d'exécutifs est l'approche la plus intuitive *a priori*, elle pose de nombreux problèmes, tant au niveau des conflits ou iniquité d'accès aux ressources qu'au niveau de la duplication d'efforts de développement. Elle ne se révèle pas être une solution générique ; elle ne fonctionne que dans certains cas particuliers, au prix de concessions et de gestion au cas par cas.

L'émulation de plusieurs paradigmes sur un unique exécutif règle les problèmes de conflits, mais cette approche ne semble guère satisfaisante pour l'évolutivité ou la modularité. Elle introduit artificiellement un paradigme privilégié, reléguant les autres au second plan. De plus, cette approche peut conduire à des performances médiocres pour les paradigmes "secondaires" en raison d'une adaptation à trop haut niveau dans la pile logicielle. Cette approche est convenable pour une utilisation occasionnelle d'un paradigme mais ne semble pas en adéquation avec une utilisation "à égalité" de plusieurs paradigmes, telle que les objets CORBA parallèles par exemple. Enfin, il faut noter que les paradigmes "secondaires" sont généralement basiques, développés sur mesure, et n'ont pas la richesse d'un paradigme implémenté par un exécutif.

Enfin la reconstruction d'exécutifs par des méthodes de composants logiciels donne des résultats prometteurs, mais n'est pas si facilement extensible ou adaptable. Le portage d'un exécutif dans ces environnements passe souvent par une réécriture complète de l'exécutif en utilisant la méthodologie fournie par la plate-forme de composants. De plus, les décompositions à grain fin souffrent de problèmes de performances et la moindre évolution demande une grande expertise pour maîtriser les assemblages complexes de dizaines de composants, la complexité étant reléguée dans l'assemblage.

Ces trois solutions permettent donc dans certains cas de disposer de plusieurs paradigmes en même temps. Pour les cas simples (pas de *multi-threading*, TCP/IP), il y a des chances raisonnables qu'une juxtaposition fonctionne, éventuellement avec une adaptation rapide "à la main". Pour des combinaisons plus avancées, l'émulation de plusieurs paradigmes sur un exécutif est suffisante quand

un exécutif est clairement "dominant", mais insuffisante pour le cas général. Les plates-formes de construction d'exécutifs, bien qu'apportant des concepts intéressants, ne permettent pas réellement de disposer de l'équivalent de plusieurs exécutifs *complets* en même temps. Elles sont intéressantes pour un prototypage, mais problématiques pour utiliser des codes existants qui supposent avoir à leur disposition par exemple une implémentation MPI, CORBA, RMI Java ou HLA complète conforme à la norme.

4.3 Conclusion

Nous avons analysé dans ce chapitre les propriétés des exécutifs par rapport à une utilisation sur une grille de calcul. Cette analyse a été menée selon deux directions : l'abstraction des ressources de communication, dite "analyse verticale" ; la capacité à offrir plusieurs paradigmes simultanément, dite "analyse horizontale".

Les infrastructures existantes ont de nombreux atouts et proposent des solutions à de nombreux problèmes rencontrés pour assurer des communications sur les grilles de calcul. De nombreuses plates-formes génériques de communication existent pour prendre en charge la gestion de l'hétérogénéité des réseaux. Les approches à base de composants semblent apporter l'adaptabilité nécessaire pour les cas qui ne sont pas pris en compte par les plates-formes génériques habituelles.

Cependant, les approches existantes souffrent de nombreuses lacunes quant à la possibilité de mettre en œuvre les modèles de programmation basés sur plusieurs paradigmes sur des ressources elles-mêmes basées sur plusieurs paradigmes. Au niveau bas, les approches *génériques* restent prisonnières d'un seul paradigme. D'une part, de telles plates-formes offrent généralement une interface abstraite du même paradigme que les ressources qu'elles utilisent. D'autre part, ces plates-formes ne prennent pas en charge toute la hiérarchie de réseaux des SAN aux WAN. Au niveau haut, les approches génériques existantes offrent une interface également selon un seul paradigme, ce qui ne semble pas suffisant pour des exécutifs variés basés sur plusieurs paradigmes.

Les approches existantes pour disposer de plusieurs paradigmes ou exécutifs en même temps consistent en général en une intégration spécifique, gérée au cas par cas, de plusieurs exécutifs. Nous pensons qu'il est préférable d'utiliser des exécutifs existants plutôt que de les redévelopper complètement spécifiquement pour la cohabitation ; une simple juxtaposition d'exécutifs existants ne fonctionne cependant pas toujours. Il nous semble donc nécessaire de prendre en compte le problème de l'utilisation de plusieurs exécutifs simultanément dès la conception d'une plate-forme de communication pour les grilles.

Deuxième partie

Un modèle de plate-forme de communication pour les grilles

Chapitre 5

Un modèle de plate-forme d'intégration d'exécutifs communicants

Sommaire

5.1 Vers un modèle d'abstraction multi-paradigme **60**
5.1.1 Abstraire au-delà de la généricité *mono-paradigme* 60
5.1.2 Études de cas : parallèle *ou* réparti 61
5.1.3 Unifier les abstractions parallèle et répartie ? 64
5.1.4 Des abstractions selon plusieurs paradigmes 65
5.1.5 Conclusion 66
5.2 Architecture pour une plate-forme de communication multi-paradigme pour les grilles **67**
5.2.1 Les niveaux d'abstraction du modèle 68
5.2.2 Vue d'ensemble de la gestion des communications 68
5.3 Intégrer différents modèles d'exécution **71**
5.3.1 Problématiques 71
5.3.2 Adaptation des modèles d'exécution 72
5.4 Conclusion **74**

Nous introduisons dans cette deuxième partie du document un modèle de plate-forme de communication pour les grilles de calcul. Ce modèle gère différents paradigmes de communication et de programmation, et autorise l'utilisation des ressources variées telles qu'on les trouve sur les grilles de calcul de façon transparente par divers exécutifs. Il est basé sur l'intégration de plusieurs exécutifs pour la mise en œuvre de modèles de programmation qui utilisent plusieurs paradigmes en même temps. Ce modèle de plate-forme, dont la conception est issue de l'analyse menée au chapitre précédent, prend en compte les problématiques de la généralisation des méthodes et ressources de communication disponibles pour tous les exécutifs, et de l'utilisation simultanée de plusieurs exécutifs. Les principaux aspects qui guident sa conception sont :

- une prise en charge d'une hiérarchie complète de réseaux, des SAN aux WAN, en les exploitant suivant leur paradigme natif, et éventuellement à l'aide de méthodes spécifiques ;
- la possibilité d'utiliser des exécutifs variés selon plusieurs paradigmes ;
- la possibilité d'utiliser plusieurs exécutifs en même temps, en combinaison arbitraire ;
- la mise en commun des méthodes de communication : tous les réseaux et toutes les méthodes de communication mis à disposition de tous les exécutifs.

Ce chapitre présente la philosophie de construction et une vue d'ensemble de l'architecture de plate-forme que nous proposons. Les chapitres suivants détaillent, niveau par niveau, l'architecture de plate-forme proposée. Dans ce chapitre qui présente l'architecture générale, nous présentons dans

une première section une extension du modèle d'abstraction classique pour prendre en compte l'aspect multi-paradigme. Nous donnons ensuite une vue globale du modèle de plate-forme de communication que nous proposons. Nous présentons enfin l'architecture de la plate-forme du point de vue des modèles d'exécution.

5.1 Vers un modèle d'abstraction multi-paradigme

Cette section étudie l'intégration des abstractions de communication au-delà de la généricité "standard", dans le but d'offrir plusieurs paradigmes en même temps à la même application sur des ressources obéissant éventuellement à un autre paradigme, en prenant en compte la portabilité sur des ressources variées.

Nous définissons au préalable les notions que nous manipulons dans cette section :

Définition 5.1 : ***interface de communication*** — Nous entendons par interface de communication un ensemble de primitives logicielles qui permettent de programmer des communications sur un réseau.

Définition 5.2 : ***abstraction de communication*** — Abstraire les ressources de communication consiste en la définition d'une interface dite *abstraite* qui existe indépendamment de toute implémentation qui fournit cette interface. La définition d'une interface abstraite a pour but de masquer les différences entre les différentes implémentations qui peuvent fournir cette interface. De ce fait, même si différentes incarnations peuvent exister, l'utilisateur de l'interface (une application, un exécutif) utilise les différentes variantes de la même façon. Nous pouvons voir l'abstraction comme un "contrat" sur la façon d'utiliser les ressources. L'*abstraction* est l'action d'abstraire les ressources en une *interface abstraite*. Par raccourci de langage, l'*abstraction* peut également désigner l'interface abstraite elle-même.

Dans cette section, par convention les figures représentent des diagrammes d'empilement des couches logicielles. Le niveau d'abstraction augmente de bas en haut. Les flèches symbolisent des composants et la relation "*utilise*".

5.1.1 Abstraire au-delà de la généricité *mono-paradigme*

L'abstraction des ressources telle qu'elle existe actuellement, présentée en section 2.1, amène la *portabilité* (définition 2.11) et la *généricité* (définition 2.12). La portabilité permet à un exécutif d'utiliser plusieurs types de réseaux en utilisant des pilotes à interface uniforme. Si l'interface de ces pilotes est conçue de façon à être réutilisable d'un exécutif à l'autre, nous obtenons la généricité. Nous proposons d'ajouter un niveau d'abstraction pour aller au-delà de la généricité telle qu'elle existe actuellement, c'est-à-dire limitée à un seul paradigme à la fois. Franchir la frontière parallèle-réparti revient à introduire deux nouvelles notions : les abstractions *multi-paradigme*, et les passerelles *trans-paradigme*.

En effet, la généricité telle qu'elle est mise en œuvre habituellement reste à l'intérieur d'un paradigme particulier. Elle est basée sur la définition d'*une* interface abstraite, de niveau intermédiaire entre les interfaces du système et les interfaces utilisées par les applications. Le choix d'une interface abstraite unique est particulièrement pertinent du point de vue de la portabilité — chaque pilote ne doit implémenter qu'une interface —, mais peut être une contrainte pour la généricité. En effet, cette unique interface revêt la plupart du temps une apparence proche de l'une de ses incarnations particulière. Ceci implique des choix cruciaux, qui peuvent être adaptés à certains cas, mais pas à tous.

Par conséquent, les abstractions des environnements génériques de communication sont généralement conçues soit pour le parallélisme, soit pour le réparti ; cette coloration se ressent sur l'interface. Par exemple, l'adressage se fait par numérotation logique des nœuds (en parallélisme), soit par un système permettant de nommer des nœuds arbitraires même sans information préalable (IP par exemple, cas du réparti). Cette coloration des abstractions restreint leur expressivité à une utilisation à l'intérieur d'un paradigme de programmation. En ce sens, le modèle d'abstraction classique peut être qualifié de *mono-paradigme* : chaque interface abstraite est adaptée à une classe d'exécutif, mais pas à tous en même temps. Dans les faits — mais peut-être pas par nécessité intrinsèque — nous constatons

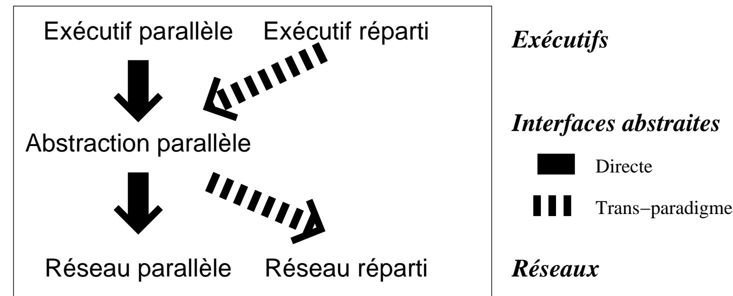

FIG. 5.1 – *Utiliser une interface abstraite parallèle en dénominateur commun.*

que le modèle d'abstraction des environnements existants est trop pauvre pour supporter en même temps plusieurs exécutifs basés sur des paradigmes différents.

Il serait souhaitable de pouvoir exprimer plusieurs paradigmes à un niveau abstrait — c'est le concept d'*abstraction multi-paradigme* — pour pouvoir utiliser simultanément plusieurs exécutifs basés sur plusieurs paradigmes. L'« analyse horizontale » a montré qu'utiliser plusieurs plates-formes simplement juxtaposées amène des conflits et ne permet pas à tous les exécutifs d'utiliser tous les réseaux. L'orientation multi-paradigme doit donc être prise en compte directement dans une plate-forme qui intègre plusieurs paradigmes.

Un autre aspect important absent du modèle d'abstraction classique est la notion de passerelle *trans-paradigme*. Nous entendons par passerelle *trans-paradigme* la possibilité d'offrir une interface abstraite de communication selon un paradigme sur un réseau conçu selon un autre paradigme. L'exemple le plus notable de passerelle trans-paradigme est MPICH-G2 [81] capable de fournir un modèle parallèle sur des ressources aussi bien parallèles que réparties. Cependant, il s'agit d'une implémentation spécifique d'un exécutif particulier ; il ne s'agit pas d'un environnement générique pour construire d'autres exécutifs. De plus, cet exemple se limite à fournir une interface de passage de messages pour le parallélisme à mémoire distribué seulement.

Dans la suite de cette section, nous étudions diverses architectures pour une plate-forme de communication multi-paradigme, capable d'offrir plusieurs paradigmes différents, en même temps, sur plusieurs types de ressources. Nous nous interrogerons en particulier sur l'interface abstraite qu'une telle plate-forme doit offrir. Le but est l'adaptation à plusieurs paradigmes et la réalisation de passerelles trans-paradigme efficaces, le tout sans pénaliser les cas où l'on n'utilise pas le caractère trans-paradigme. Nous étudierons successivement les cas d'une abstraction unique parallèle ou répartie, d'une abstraction unifiée entre les deux paradigmes, et une abstraction hybride, présentant les deux types d'interfaces.

5.1.2 Études de cas : parallèle *ou* réparti

Dans cette section, nous envisageons le cas où une plate-forme de communication présente une interface abstraite obéissant soit au paradigme parallèle, soit au paradigme réparti. Nous étudions l'adéquation d'un tel modèle d'abstraction pour offrir des exécutifs selon plusieurs paradigmes de communication en même temps sur des ressources pouvant être orientées selon plusieurs paradigmes, y compris les passerelle *trans-paradigme*.

5.1.2.1 Abstraction unique de type parallèle à mémoire distribuée

Dans un premier temps, nous étudions le cas d'une plate-forme de communication basée sur une interface abstraite unique de type parallèle à mémoire distribuée. Cette approche est symbolisée par la figure 5.1. Dans ce cas, une interface abstraite pour le parallélisme joue le rôle de *dénominateur commun* à toutes les communications sur la grille, à l'aide d'une seule API. C'est une approche semblable à celle de la section 3.2, mais à un niveau abstrait ; alors que l'approche présenté en section 3.2 consiste en la

programmation d'une grille à l'aide d'un seul exécutif pour le parallélisme, nous étudions ici le cas de l'utilisation de plusieurs exécutifs construits sur une seul interface abstraite pour le parallélisme.

Une telle interface abstraite peut être par exemple Madeleine [8], qui permet de programmer des grappes de grappes avec Madeleine 3. Dans les faits, nous constatons que les utilisateurs de grilles de calcul utilisent souvent MPICH-G2 [81] ou une autre implémentation MPI à grande échelle telle que l'implémentation MPI d'Albatross [119] (à savoir : MAGPIE [120]) conçue pour des grappes de grappes à grande échelle ou encore PACX-MPI [49] dont le but est d'interconnecter plusieurs supercalculateurs. Même si MPI est plutôt une interface destinée à être utilisée par une application et n'est pas une interface *générique* de communication destinée à être utilisés par des exécutifs, il nous semble malgré tout pertinent d'étudier le cas où MPI est utilisée comme interface abstraite pour les grilles, dans la mesure où cela correspond à la réalité de son utilisation. Nous nous intéressons à tous les cas représentés par la figure 5.1, approche centrée sur une abstraction parallèle avec d'inévitables passerelles pour passer d'un paradigme à l'autre lors de l'utilisation de réseaux ou d'exécutifs répartis.

Pour les exécutifs parallèles, cette façon de faire ne semble pas poser de problème particulier ; même sur des réseaux de type "réparti", cette approche est viable dans la mesure où il est nécessaire de réaliser une passerelle trans-paradigme de toute façon.

Pour des exécutifs répartis, en revanche, cela semble plus problématique, et particulièrement dans le cas d'une utilisation sur des réseaux de type réparti. En effet, le passage par une unique interface de type parallèle perd des propriétés essentielles du réparti utilisées par les exécutifs répartis, tels que CORBA, SOAP ou HLA. Ceci n'est pas lié à une implémentation particulière d'un quelconque environnement générique parallèle mais à l'abstraction parallèle elle-même : une interface abstraite parallèle possède une expressivité restreinte pour décrire certaines notions essentielles du calcul réparti.

La différence entre les notions manipulées par une interface abstraite pour le parallélisme ou pour le réparti transparaissent de façon intuitive au niveau de l'API. Une abstraction parallèle utilise une interface de programmation spécifique différente d'une API répartie. Par exemple, pour les mécanismes de connexion, les environnements parallèles construisent la topologie en début de session ; les primitives `bind`, `accept` ou `connect` du réparti n'ont pas d'équivalent en parallèle. Par conséquent, une implémentation SOAP qui chercherait à écouter spécifiquement sur le port 80 dédié à HTTP ne pourrait pas exprimer une telle requête à l'aide d'une interface abstraite pour le parallélisme [1]. Plus précisément, les notions du réparti perdues par une interface abstraite unique pour le parallélisme sont :

Mécanismes d'adressage — En parallélisme à mémoire distribuée, l'adressage est basé sur la notion de *monde* qui comprend un certain nombre de nœuds. Il est possible de connaître le nombre de participants dans le *monde*. La désignation des nœuds se fait par numérotation logique, généralement de 0 à $n-1$ pour un monde de taille n. En réparti, l'adressage est basé sur les adresses IP, avec éventuellement l'utilisation de noms symboliques résolus ensuite en adresse IP. Il s'ensuit qu'un exécutif réparti qui utilise une abstraction parallèle ne peut adresser que des nœuds déjà connus (à l'intérieur du *monde*) par l'intermédiaire de leur numéro ; à cause de l'interposition d'une abstraction parallèle, un tel exécutif a perdu la capacité d'adresser des nœuds qui sont en dehors du *monde* car leur adresse n'est pas exprimable dans le système parallèle.

Topologie et dynamicité — En réparti, la topologie est entièrement dynamique. La structure de base est le *lien*, établi dynamiquement à l'aide de primitives de type `connect` et `accept`. En parallélisme, la structure logique essentielle est une *clique* (graphe où tous les nœuds sont connectés logiquement à tous les autres) sur un ensemble de nœuds déterminé et statique. Dans certains cas, la topologie n'est pas statique (MPI-2, PVM, prochaines versions de Madeleine) ; cependant, il s'agit de changements dynamiques dans la topologie du *monde* en lançant des nœuds supplémentaires, ce qui est différent du lancement de nœuds indépendants puis d'un établissement de connexion entre ces deux nœuds à l'initiative du client.

Interopérabilité — Les exécutifs répartis définissent pour la plupart des protocoles d'interopérabilité pour pouvoir se connecter à des implémentations différentes de l'exécutif. C'est un aspect essentiel du réparti pour gérer l'hétérogénéité : les interactions se basent sur des conventions accep-

1. à moins d'étendre spécifiquement l'interface dans cette direction, ce qui revient au cas étudié dans la section 5.1.3.

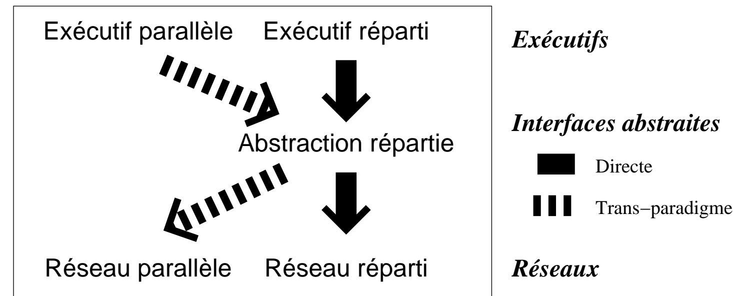

FIG. 5.2 – *Utiliser une interface abstraite répartie en dénominateur commun.*

tées par tous comme langue commune. Il s'agit par exemple des protocoles IIOP pour CORBA ou HTTP pour SOAP. À l'inverse, un environnement générique pour le parallélisme a toute liberté pour utiliser son propre protocole de transmission car il ne communique qu'avec des nœuds connus qui utilisent la même implémentation. Ni MPI ni Madeleine, même sur TCP/IP, ne sont susceptibles de recevoir des messages IIOP ou HTTP standard ; ce n'est d'ailleurs pas leur but.

L'utilisation d'une unique abstraction de communications de type parallèle pour toutes les communications et pour tous les exécutifs apporte donc des limitations aux exécutifs réparti. Nous constatons qu'un exécutif réparti qui utilise un réseau réparti au travers d'une interface abstraite parallèle n'est pas équivalent au même exécutif qui utilise le même réseau réparti directement ou par l'intermédiaire d'une interface abstraite répartie. L'abstraction parallèle ne permet pas d'exprimer certaines propriétés essentielles du réparti. Ces propriétés présentes au niveau du réseau et utilisées au niveau de l'exécutif sont perdues par l'interface abstraite parallèle. En particulier, on notera que cette approche ne permet pas de réaliser des communications CORBA ou SOAP conformes aux normes qui soient interopérables avec leur version standard. Cette solution n'est donc pas satisfaisante.

5.1.2.2 Abstraction unique de type réparti

Après avoir étudié le cas d'une unique abstraction parallèle à mémoire distribuée pour utiliser les réseaux d'une grille, nous étudions ici le cas d'une plate-forme de communication basée sur une unique abstraction répartie. Cette approche est illustrée à la figure 5.2.

L'idée qui vient immédiatement à l'esprit quand il s'agit d'interconnecter des ressources variées à large échelle consiste à se reposer sur le *protocole internet* (IP) [150]. IP est alors considéré comme un dénominateur commun de la grille, permettant de réaliser toutes les communications. Vue l'omniprésence d'IP, cette approche a de grandes chances de fonctionner. Les contraintes d'adressage, de dynamicité de la topologie et d'interopérabilité demandée par les exécutifs répartis sont remplies. D'un point de vue purement fonctionnel, il est envisageable également de faire fonctionner des exécutifs parallèles sur IP. Il faut d'ailleurs remarquer que la plupart des plates-formes génériques pour le parallélisme telles que Madeleine, Panda, ou Ibis existent sur TCP/IP. La plupart du temps, l'ensemble peut donc indubitablement fonctionner avec une interface abstraite commune de type réparti. Sans se limiter à IP, d'autres abstractions réparties sont envisageables, par exemple ADAPTIVE [164] ; les priorités restent les mêmes : interopérabilité et topologie gérée lien par lien.

Cependant, une approche qui ramène toutes les communications à une interface abstraite de type réparti perd de vue l'essence-même du parallélisme, qui est la recherche de haute performance. Les abstractions parallèles sont d'ailleurs en fait des abstractions distribuées spécialisées pour tenir compte des priorités du parallélisme. Les contraintes plus relâchées du parallélisme en ce qui concerne l'interopérabilité ou la sécurité sont concédées dans le but de permettre des simplifications et optimisations pour obtenir une meilleure performance. Contraindre les exécutifs parallèles à se baser sur une interface de type réparti revient à renoncer à toutes les spécificités du parallélisme, à savoir : des API de communication optimisées, des opérations collectives qui tiennent compte de la topologie matérielle, et d'une manière générale une conception guidée par la haute performance plutôt que par

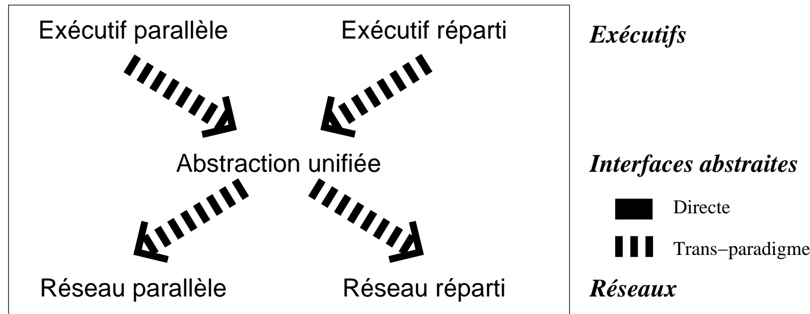

FIG. 5.3 – *Utiliser une interface abstraite unifiée.*

l'interopérabilité.

Si la majorité des constructeurs de machines parallèles fournissent une implémentation MPI ou, à défaut, une bibliothèque orientée "parallélisme", la possibilité d'utiliser le réseau interne de ces machines par IP est plus rare, et la plupart du temps offre des performances médiocres. Dans les cas où l'implémentation IP existe sur toutes les machines et réseaux visés, c'est une solution qui fonctionne et permet de déployer des codes aussi bien parallèles que répartis sur l'ensemble des ressources. Cependant, se restreindre à IP sans tirer profit des capacités spécifiques au parallélisme offertes par les machines parallèles diminue l'intérêt d'acheter de telles machines. Beaucoup de travaux sur les environnements de communication pour le parallélisme ont été menés précisément dans le but de court-circuiter TCP/IP !

5.1.3 Unifier les abstractions parallèle et répartie ?

La section précédente a mis en évidence que ni une abstraction purement orientée parallèle à mémoire distribuée ni une abstraction purement orientée réparti ne sont satisfaisantes pour que des exécutifs basés sur des paradigmes différents puissent utiliser efficacement différents réseaux. Chacune des deux abstractions a des lacunes qui empêchent d'exprimer certaines notions indispensables ou prioritaires pour l'autre paradigme. Les approches "répartie" et "parallèle" des interfaces abstraites de communication ne sont que deux cas particuliers conçus pour des utilisations spécifiques. Il n'est donc guère surprenant qu'elles ne conviennent pas aux cas pour lesquels elles n'ont pas été conçues. Il serait donc intéressant de chercher une nouvelle abstraction de communication orientée "*grille*" qui permettrait d'exprimer l'*union* des notions des deux paradigmes, comme l'illustre la figure 5.3. Est-il envisageable de trouver une telle abstraction qui combine les bonnes propriétés des deux paradigmes ?

Nous étudions cette question selon deux axes : les mécanismes d'adressage et ceux d'échange de messages.

Adressage et topologie. Les exécutifs répartis tels que CORBA ou SOAP sont basés sur le principe de connexion dynamique à des nœuds dont on ne connaît rien d'autre *a priori* que leur adresse IP et un numéro de port. Les informations de topologie nécessaires sont restreintes au seul lien concerné pour faciliter la dynamicité et l'interopérabilité. À l'inverse, les exécutifs parallèles sont basés majoritairement sur le principe SPMD et une clique de communication, donc une connaissance globale de la topologie. Quand la dynamicité existe, elle se limite à un lancement par l'exécutif lui-même (*spawn* de PVM ou MPI-2) de façon à faciliter la connaissance de la topologie ; ce n'est pas l'interconnexion arbitraire de nœuds déjà lancés de façon autonome, ce qui remettrait en cause la topologie globale. Cette connaissance globale de la topologie d'interconnexion des nœuds est essentielle pour les optimisations des opérations collectives [116, 174].

Une gestion de topologie unifiée entre parallélisme et réparti devrait de toute façon faire le choix entre la propagation des informations de topologie à un ensemble de nœuds déterminé, ou une connaissance restreinte à un lien à la fois. L'interopérabilité requise par l'aspect "réparti" limite la

connaissance de la topologie manipulée par les primitives de connexion au lien seul, ce qui n'est pas satisfaisant pour une utilisation par un exécutif parallèle.

Échange de messages. La priorité pour l'échange de message en parallélisme est la performance, alors qu'en réparti l'interopérabilité est prioritaire, même si la performance n'en est pas pour autant négligée. Cette différence se traduit au niveau des primitives d'envoi et réception. La plupart des protocoles du réparti sont construits sur TCP avec des flux continus ; les réseaux et exécutifs du parallélisme utilisent des messages bien délimités. Il n'est fondamentalement pas impossible de trouver une interface unique qui réalise toutes les fonctionnalités demandées. Il est en effet très facile d'utiliser une interface basée sur les flux pour envoyer des messages délimités, la transformation entre flux et messages délimités pouvant être réalisée plusieurs fois dans la pile de protocole.

Cependant, les efforts existants actuellement sur les interfaces d'échange de messages en parallélisme ont pour but de permettre une bonne expressivité des besoins de l'exécutif ou de l'application pour réaliser au niveau inférieur les optimisations en conséquence, sans transformation superflue. C'est le cas par exemple des interfaces à construction incrémentale des messages (`pack/unpack` dans Madeleine et PVM) qui permettent une transmission optimisée des messages de longueur variable typiques du LRPC.

Bilan : faisabilité et pertinence de l'approche unifiée. Une interface abstraite unifiée entre parallélisme et réparti est sans aucun doute faisable. Cependant, notre étude a montré qu'une telle interface ne peut pas être très éloignée du modèle réparti, principalement par la nécessité d'interopérabilité. Or, il ne semble pas souhaitable de tout ramener au réparti car cela revient à renoncer aux particularités du parallélisme. Ceci complique du même coup les optimisations apportées par le relâchement des contraintes des modèles parallèles, voire les rendrait impossibles. Au regard de ces considérations, il semble donc que chercher à tout prix une interface abstraite unifiée entre calcul réparti et parallélisme revient à être borné par les contraintes les plus fortes (celles du réparti) et à renoncer à l'essentiel des optimisations possibles et souhaitables dans les cas où les contraintes sont plus relâchées (en parallélisme). La question réelle ne concerne donc pas la *faisabilité* d'une approche unifiée mais sa *pertinence*.

Une telle interface abstraite est utilisée par des exécutifs qui sont soit répartis, soit parallèles, chacun avec ses propres besoins et préférences. Or une abstraction unifiée regroupe non seulement les propriétés mais aussi les contraintes : si une préférence d'un paradigme est contraire aux besoins d'un autre paradigme, alors il faut respecter le besoin impératif au détriment de la préférence. Cette contrainte n'est contrebalancée que par un seul avantage : la définition d'une seule interface au lieu de deux. Cet avantage ne pèse pas beaucoup dans la mesure où chaque exécutif qui utilise une interface abstraite l'utilise soit comme une abstraction parallèle, soit comme une abstraction répartie. Il n'y a à notre connaissance pas d'exécutif qui mette à profit les aspects réparti et parallèle en même temps. La définition d'une interface abstraite unifiée ne nous semble donc pas pertinente.

5.1.4 Des abstractions selon plusieurs paradigmes

Nous proposons dans cette section un modèle d'abstraction qui ne fait pas de compromis superflus, en proposant plusieurs interfaces abstraites, chacune adaptée à un paradigme.

Le point commun entre les différents cas de figure étudiés pour une abstraction unique — parallèle, répartie, ou unifiée — pour construire des exécutifs selon plusieurs paradigmes est le conflit entre les contraintes et préférences des différents paradigmes. L'utilisation d'une seule abstraction mène à une perte de propriétés importantes si l'abstraction est trop restrictive, ou amène à des compromis sinon. Par exemple, utiliser uniquement une abstraction parallèle, même sur un réseau réparti, perd la propriété d'interopérabilité dont un exécutif réparti pourrait avoir besoin. De même, une abstraction répartie ne permet pas d'exprimer des propriétés d'optimisation utilisées potentiellement par un exécutif parallèle sur un réseaux parallèle au travers d'une abstraction répartie. Une interface intermédiaire unique amène des compromis en cherchant à exprimer des propriétés contradictoires en une seule interface.

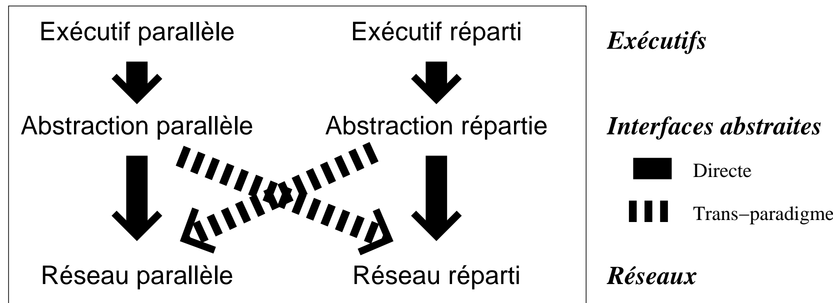

FIG. 5.4 – *Utiliser différentes interfaces abstraites pour différents paradigmes*

Ne faire que les *compromis nécessaires.* Il nous semble plus pertinent de proposer chacune des interfaces abstraites aux exécutifs. Les concepteurs de chaque exécutif peuvent choisir l'interface abstraite qui convient le mieux, selon le paradigme. Chaque abstraction est incarnée sur chaque type de réseau, comme illustré à la figure 5.4. Cette approche n'impose pas d'interface unique, donc aucune contrainte globale. De cette façon, les propriétés qui ne sont pas exprimables dans toutes les abstractions ne sont pas perdues. Chaque adaptation pour proposer une interface selon un paradigme sur un type de réseau peut être réalisée *au mieux*, indépendamment de toute interface imposée.

Ceci signifie qu'il faut réaliser plusieurs incarnations de chaque abstraction. Avec les paradigmes parallèle et réparti, ceci signifie : abstraction parallèle sur réseau parallèle, abstraction répartie sur réseau réparti, abstraction parallèle sur réseau réparti et abstraction répartie sur réseau parallèle ; les deux premières sont qualifiées de *directes* alors que les deux dernières sont des passerelles *trans-paradigme*. Les incarnations directes ne font aucun compromis et ne perdent pas de propriétés du paradigme. Les incarnations trans-paradigme ne réalisent que les compromis nécessaires : les compromis pour passer d'un paradigme à l'autre ne sont fait effectivement que quand on demande une abstraction sur un réseau d'un autre paradigme.

Limites de l'intégration de plusieurs abstractions. Par rapport à une portabilité classique par pilotes, nous sommes encore en présence d'une multiplication du nombre de combinaisons à implémenter. En revanche, cette multiplication ne porte pas sur le nombre d'exécutifs ni de réseaux, mais seulement sur le *nombre d'abstractions*. Pour disposer d'une abstraction de chaque type sur des réseaux basés sur n paradigmes différents, il faut réaliser n^2 incarnations. Nous avons considéré jusqu'à présent dans notre étude que ce nombre était limité à 2 (réparti et parallèle à mémoire distribuée). Ajouter le paradigme parallèle à mémoire partagée fait passer le nombre de paradigmes à 3 et semble faire passer le nombre d'incarnations à 9. Nous verrons par la suite qu'il n'en est rien car les parallélismes à mémoire distribuée ou partagée ont beaucoup en commun. De plus, les adaptations "directes" (3 parmi les 9) sont triviales. Le nombre de paradigmes à gérer étant restreint, le nombre de combinaisons à prendre en compte reste raisonnable.

5.1.5 Conclusion

Nous avons étudié dans cette section une extension du modèle d'abstraction classique des ressources de communication, de façon à prendre en compte le besoin de multiples exécutifs basés sur différents paradigmes simultanément. Nous avons montré qu'il est nécessaire de prendre en compte le caractère *multi-paradigme* et les passerelles entre paradigmes au niveau des abstractions, c'est-à-dire en dessous des exécutifs plutôt qu'au-dessus. Nous avons étudié les cas d'utilisation d'une abstraction unique par différents types d'exécutifs, et montré que chaque cas — réparti ou parallèle — n'est pas satisfaisant car ne permet pas d'exprimer toutes les propriétés souhaitées. Nous avons envisagé une unification entre les abstractions parallèle et répartie, et nous avons déduit que cette approche est faisable mais ne semble pas pertinente car elle résulte en des compromis qui ne sont pas nécessaires. Enfin, nous avons proposé d'intégrer plusieurs abstractions — une abstraction pour chaque

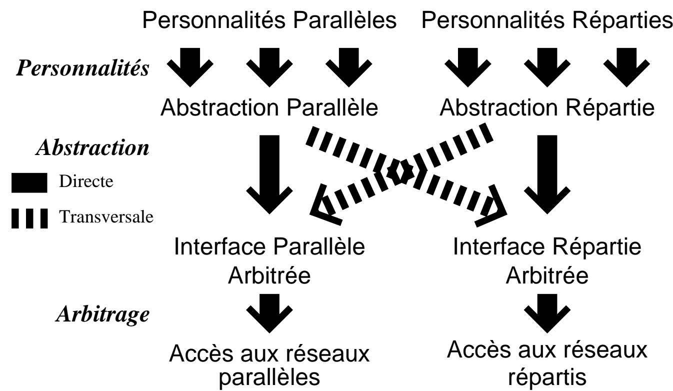

FIG. 5.5 – *Modèle proposé avec les paradigmes parallèle à mémoire distribuée et réparti.*

paradigme —, et de réaliser les passerelles *trans-paradigme* sous les abstractions, uniquement dans les cas où cela est nécessaire. La force de ce dernier modèle est de ne contraindre aucune interface intermédiaire à un paradigme déterminé, ce qui élimine les compromis superflus amenés par le choix arbitraire d'une interface unique. Nous proposons donc d'utiliser ce modèle d'intégration d'abstraction pour réaliser une plate-forme générique de communication pour les grilles.

5.2 Architecture pour une plate-forme de communication multi-paradigme pour les grilles

Cette section présente l'architecture générale d'un modèle de gestion des communications dans une plate-forme pour les grilles. Cette architecture est construite autour du modèle d'abstraction multi-paradigme étudié à la section précédente. La présentation de cette architecture s'articule en deux parties : d'abord la définition des niveaux d'abstraction considérés dans ce modèle, puis la présentation des différentes couches logicielles correspondant à ces niveaux et leur rôle. Les caractéristiques principales de ce modèle sont :

multi-paradigme — Ce modèle est multi-paradigme dans le sens où il propose plusieurs abstractions de communications. La même plate-forme peut servir pour des exécutifs qui obéissent à différents paradigmes.

multi-exécutif — Cette architecture est conçue de façon à faire cohabiter plusieurs exécutifs simultanément sur la même plate-forme de communication.

flexible — Pour un maximum de souplesse d'utilisation et une adaptation à des conditions variées,cette architecture de plate-forme de communication repose sur des éléments composables au choix selon les besoins. Chacun de ces éléments — chaque "brique" de base — est un composant, entité encapsulée aux interfaces clairement identifiées, appelées *ports*, et composables par une tierce partie.

performant — La haute performance est assurée à tous les niveaux ; les abstractions et empilement logiciels ne pénalisent pas la performance.

La figure 5.5 présente un exemple de l'utilisation dans une plate-forme complète de l'extension du modèle d'abstraction que nous avons présenté dans la section précédente. En plus d'une couche d'abstraction multi-paradigme, une telle plate-forme met en œuvre un arbitrage en-dessous du niveau d'adaptation d'abstraction, et un mécanisme de personnalité au-dessus, dans le but de supporter plusieurs exécutifs *simultanément*. La section suivante décrit ces niveaux et le modèle d'un point de vue général.

5.2.1 Les niveaux d'abstraction du modèle

L'architecture du modèle est organisée suivant plusieurs niveaux ; chaque niveau correspond à un niveau d'abstraction — et donc de service — déterminé. À chaque niveau, il peut exister plusieurs composants qui assurent un service de même niveau d'abstraction, mais pour des paradigmes différents, des méthodes de communication différentes, ou avec des implémentation différentes. Notre modèle est une extension du modèle d'abstraction des ressources classique présenté à la section 2.1.3.2. Les différents niveaux sont illustrés par la figure 5.6. Nous présentons ici l'intégralité des niveaux de notre modèle, par niveau d'abstraction croissant (de bas en haut).

système — Le niveau *système* est implémenté sous la forme d'un pilote, tel que BIP, IB Access ou SISCI, et peut être le cas échéant inclus dans le noyau. Le but de ce niveau est de fournir une API primitive pour accéder au matériel. L'API peut éventuellement être différenciée selon le type de matériel.

portabilité, généricité — Le niveau de portabilité a pour but de cacher la différence d'API entre les pilotes lors de l'accès aux réseaux. Comme posé par la définition 2.12 page 14, nous parlons de généricité quand l'infrastructure logicielle qui assure la portabilité n'est pas spécifique à un exécutif particulier. À ce niveau, nous pouvons citer Madeleine, Panda, Globus XIO, ADAPTIVE.

arbitrage — L'arbitrage apporte la réentrance, le multiplexage, et une harmonisation des choix globaux. Le but de l'arbitrage est de résoudre les conflits d'accès aux ressources qui surgissent quand plusieurs exécutifs et/ou applications demandent l'accès au réseau en même temps, comme présenté à la section 4.2.2. L'API d'un service arbitrée est similaire à l'API du niveau générique, à la différence que plusieurs codes différents peuvent utiliser les ressources en même temps.

adaptation d'abstraction — Le niveau d'adaptation d'abstraction a pour but d'offrir chaque abstraction choisie (parallèle et réparti, par exemple) sur chaque ressource disponible. C'est à ce niveau qu'a lieu la traduction trans-paradigme.

virtualisation — Le niveau de virtualisation adapte l'API abstraite en plusieurs personnalités susceptibles d'être utilisées par les exécutifs.

exécutif — Les exécutifs implémentent un modèle de programmation particulier. À ce niveau, nous pouvons citer CORBA, MPI, RTI HLA, *Web Services*. Leur API "standard" est destinée à être utilisée par les applications.

application — Le niveau applicatif regroupe tous les codes au-dessus du niveau des exécutifs. Il peut s'agir de codes écrits par l'utilisateur final comme de bibliothèques de calcul spécialisées.

Les interconnexions entre les composants des différents niveaux est représentée à la figure 5.6 en notations orientées "composants" ; nous utiliserons désormais ces notations dans la suite de ce document. Sur l'exemple de la figure, trois types de réseaux sont disponibles ; il sont abstraits en deux paradigmes distincts, à l'aide des adaptateurs adéquats ; chaque abstraction est proposée sous trois personnalités différentes. Ces notions sont définies dans les sections suivantes. Cet exemple est donné pour illustrer la généralité du modèle. Dans la suite, nous considérerons essentiellement des cas à deux ou trois paradigmes et réseaux.

5.2.2 Vue d'ensemble de la gestion des communications

Dans cette section, nous présentons un modèle de gestion des communications basé sur les niveaux définis à la section précédente et présentés sur la figure 5.6. La plate-forme de communication en elle-même est constituée des trois couches centrales, à savoir : arbitrage, adaptation d'abstraction, virtualisation. Les autres niveaux sont soit utilisés par la plate-forme, soit utilisateurs de la plate-forme. Nous décrivons dans l'ordre les couches logicielles d'arbitrage, d'adaptation d'abstraction, et de virtualisation.

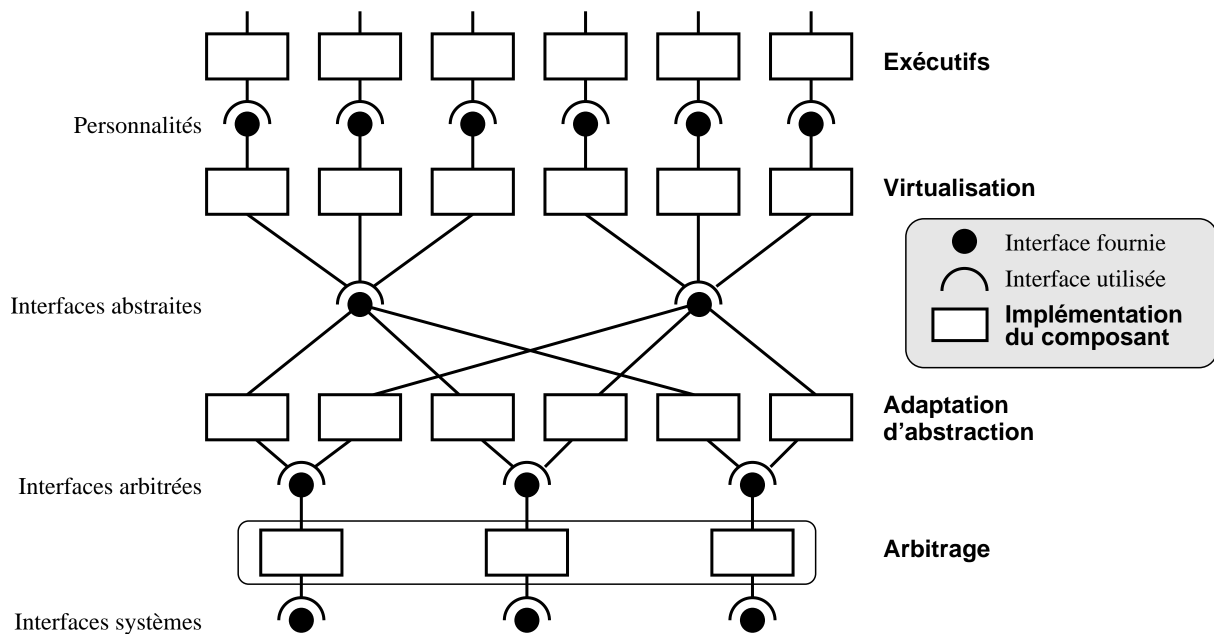

FIG. 5.6 – *Vue générale de l'architecture de plate-forme de communication proposée.*

5.2.2.1 Arbitrer

La couche la plus basse de la plate-forme assure l'arbitrage lors de l'accès au ressources. Les méthodes d'accès aux réseaux — environnement générique ou API du système — ont certaines contraintes et limitations ; de plus, elles ne sont pas conçues pour fonctionner ensemble. La couche d'arbitrage apporte un support de la concurrence au-dessus de ces méthodes d'accès. La concurrence a lieu entre des accès multiples au même réseau et entre des accès à différents réseaux.

Le principe d'une plate-forme générique de communication proposant plusieurs paradigmes permet de n'utiliser qu'une seule plate-forme pour plusieurs exécutifs en même temps. De fait, la plupart des problèmes de cohabitation décrits à la section 4.2.2 sont reportés dans la plate-forme elle-même. La gestion de tels problèmes directement dans la plate-forme est alors grandement facilitée.

Les fonctionnalités ajoutées par l'arbitrage au-dessus des méthodes d'accès aux réseaux sont :

réentrance — La réentrance est la capacité pour une interface d'être utilisée par plusieurs clients en même temps. Il s'agit d'ajouter des protections par exclusion mutuelle.

multiplexage — Le multiplexage est la capacité de faire passer plusieurs connexions logiques sur le même lien physique.

coopération de scrutation — Nous proposons une coopération entre les différents mécanismes de scrutation du réseau pour éviter la compétition qui peut aller jusqu'à l'interblocage. Pour gérer au mieux la collaboration d'accès, la couche d'arbitrage gère elle-même la scrutation de tous les réseaux.

priorité et équité — Plusieurs accès aux réseaux lancés de façon indépendante entrent en compétition. Nous proposons des mécanismes qui assurent l'équité entre les différents accès simultanés.

harmonisation et virtualisation — Certaines actions ont un effet global à l'intérieur d'un processus. Nous harmonisons les choix globaux qui peuvent l'être, et rendons locaux les autres par l'intermédiaire de mécanismes de virtualisation.

Pour assurer ces propriétés, la couche d'arbitrage met en œuvre un *scrutateur* permanent qui assure une faible latence pour les diverses opérations. Le scrutateur permet de reporter les décisions d'ordonnancement des accès au réseau habituellement prises à haut niveau dans une couche plus basse, de façon à gérer les interaction entre réseaux de façon cohérente. Un scrutateur est affecté à chaque canal d'accès au réseau. Cette architecture prévoit de plus un contrôle d'interaction entre les scruta-

teurs. Chaque scrutateur peut être doté d'un *multiplexeur/démultiplexeur* si le réseau ne dispose pas de multiplexage natif, et contrôler la réentrance si la méthode d'accès n'est pas réentrante.

Le chapitre 8 est consacrée à la description de ces mécanismes d'arbitrage.

5.2.2.2 Adapter les abstractions

La couche de niveau intermédiaire dans la plate-forme de communication réalise l'adaptation des abstractions. Elle propose une interface abstraite de chaque paradigme sur chacune des interfaces de la couche d'arbitrage. C'est dans la couche d'adaptation d'abstraction qu'a lieu la traduction trans-paradigme lorsqu'elle est nécessaire. Elle offre ainsi une abstraction unifiée pour chaque paradigme :

– une abstraction pour le parallélisme à mémoire distribuée ;
– une abstraction pour le calcul réparti ;
– une abstraction pour le parallélisme à mémoire partagée.

Ce niveau est construit entièrement de façon modulaire. L'élément de base de l'adaptation d'abstraction est un composant appelé *adaptateur*. Le but d'un adaptateur est de *fournir* une interface abstraite de communications. La plupart du temps, il *utilise* une interface de la couche d'arbitrage pour fournir ses services. Dans des cas avancés, il peut également utiliser d'autres adaptateurs ou toute autre ressource jugée utile pour offrir son service.

Par le jeu de l'*assemblage* des différents adaptateurs au-dessus de la couche d'accès arbitré, la couche d'adaptation d'abstraction offre un accès selon le paradigme voulu à toutes les ressources disponibles, en pouvant utiliser toutes les méthodes de communications. La question cruciale du *choix* de l'adaptateur ou de l'assemblage à utiliser — et donc de la méthode de communications à employer — est confiée à une entité appelé *sélecteur*. Le sélecteur prend ces décisions en fonction de sa connaissance de la topologie du réseau, des capacités des adaptateurs, et de paramètres de configuration choisis par l'utilisateur.

Le chapitre 6 est consacrée à la couche d'adaptation d'abstraction.

5.2.2.3 Virtualiser les interfaces

La couche de plus haut niveau dans la plate-forme de communications assure la virtualisation de l'interface de programmation. Son rôle est d'adapter l'interface générique présentée par la couche d'adaptation d'abstraction à l'interface demandée par chacun des exécutifs qui utilisent la plate-forme de communication.

La virtualisation consiste en l'émulation d'une ressource en utilisant une autre ressource. La couche de virtualisation émule l'interface et le comportement de la ressource que chacun des exécutifs utilise habituellement ; cette émulation est réalisée au-dessus des interfaces génériques de la couche d'adaptation d'abstraction. De cette façon, un exécutif qui utilise l'interface virtuelle utilisera en fait, "sans s'en apercevoir", la plate-forme de communication. Ceci apporte les possibilités d'utiliser divers réseaux et méthodes de communication offertes par la couche d'adaptation d'abstraction à des exécutifs sans avoir besoin de les modifier.

Les exécutifs utilisant des interfaces variées pour accéder au réseau, il est nécessaire de prévoir plusieurs interfaces virtuelles différentes au-dessus des interfaces génériques. Ces différentes interfaces sont des *personnalités*. Chaque personnalité adapte l'API pour offrir l'API attendue par les exécutifs. Cependant, les personnalités adaptent uniquement l'API ; elles ne réalisent pas de transformation de paradigme.

Les mécanismes de virtualisation par personnalités sont un moyen de mettre à la disposition de tous les exécutifs, sans les modifier, les possibilités offertes par les couches d'adaptation d'abstraction et d'accès arbitré. Ainsi, il est possible d'utiliser des exécutifs sur des réseaux pour lesquels ils n'ont pas été conçus, et en même temps que d'autres exécutifs dans le même processus.

La virtualisation d'interface par personnalité fait l'objet du chapitre 7.

5.3 Intégrer différents modèles d'exécution

La section précédente a introduit un *modèle de communication* multi-paradigme. Nous introduisons dans cette section un *modèle d'exécution* multi-paradigme. Nous examinons les implications des propositions de modèle d'abstraction multi-paradigme et de la virtualisation sur les modèles de déploiement et d'exécution. Nous nous focaliserons tout particulièrement sur les passages trans-paradigme et l'utilisation de plusieurs exécutifs simultanément.

5.3.1 Problématiques

La plupart des exécutifs sont associés à un modèle d'exécution particulier : MPI est associé au modèle SPMD, CORBA, ICE et SOAP au client/serveur, HLA à une fédération (dynamique ou statique, selon les implémentations), etc. Les différences entre les modèles d'exécution peuvent être définies par certaines propriétés :

dynamicité — il s'agit de la capacité à établir ou couper des connexions alors que l'application est en cours d'exécution. C'est par exemple la façon de faire habituelle en CORBA ou SOAP, impossible en MPI-1, évènement occasionnel en MPI-2 ou PVM.

synchronisation — certains modèles fournissent une synchronisation des différents nœuds, en particulier en SPMD les processus démarrent de façon synchronisée. D'autres modèles au contraire fonctionnent sans aucune synchronisation entre les processus.

connaissance des ressources — certains modèles se basent sur la connaissance des ressources sur lesquelles l'application est exécutée. Ainsi, la liste des machines à utiliser par une application MPI doivent être listées à l'initialisation. À l'inverse, des modèles tels que ceux proposés par SOAP ou CORBA peuvent utiliser des ressources totalement arbitraires à partir du moment où elles peuvent être désignées par leur adresse IP. RMI Java adopte une démarche intermédiaire en nécessitant l'utilisation d'un registre de connaissance (RMI *registry*) où les ressources sont enregistrées dynamiquement.

Ce lien qui existe entre chaque exécutif et un modèle d'exécution associé facilite la mise en œuvre en utilisation "normale", c'est-à-dire pour une application basée sur un exécutif, qui utilise son modèle d'exécution associé et est déployée sur des ressources conçues pour le paradigme correspondant. Dans les paragraphes suivants, nous décrivons deux cas où ce lien imposé risque d'être une limitation : pour un déploiement sur des ressources répondant à un autre paradigme que celui de l'exécutif, et lors de l'utilisation de plusieurs exécutifs en même temps, chacun fournissant son propre modèle d'exécution.

Modèle d'exécution trans-paradigme. Lors de l'utilisation d'un adaptateur *trans-paradigme*, un exécutif est amené à utiliser des ressources réseaux pour lesquelles il n'a pas été conçu. Or il est courant que les ressources réseaux soient conçues en fonction des exécutifs typiquement utilisés.

Par exemple les réseaux que nous avons classés comme "parallèles" peuvent utiliser un mécanisme de lancement de processus spécifique, adapté aux exécutifs parallèles : souvent un mécanisme statique, avec synchronisation sur un ensemble de nœuds déterminé et connu à l'avance. Lors de l'utilisation d'un adaptateur de type réparti/parallèle, ces mécanismes peuvent se révéler inadaptés pour déployer un exécutif de type réparti. Ces mécanismes prennent souvent la forme d'un *lanceur* qui bâti la topologie en début de session grâce à une liste de machines fournie. Par exemple l'utilisation de Myrinet à l'aide de BIP passe par le lanceur `bipload`, la plupart des réseaux internes de supercalculateurs demandent de passer par le `mpirun` fourni par le constructeur ou un équivalent si l'on utilise une interface de passage de messages de plus bas niveau spécifique à la machine, enfin les bibliothèques génériques de communications pour le parallélisme fournissent leur propre lanceur en frontal des lanceurs spécifiques aux réseaux ; par exemple l'utilisation de Madeleine passe par `pm2load` ou `leonie`. Il est clair qu'il est alors difficile de faire fonctionner des exécutifs tels que CORBA, SOAP, RMI Java ou HLA pour lesquels la durée de vie des différents processus impliqués n'est pas la même. Par exemple en CORBA, quand un serveur est lancé, il peut servir de nombreux client qui se connectent et se déconnectent dynamiquement, la plupart des clients ayant une durée de vie beaucoup plus courte que

le serveur. De plus, il est possible de lancer un client sur un nœud qui n'était pas connu au moment du lancement du serveur, alors que les lanceurs pour les différents réseaux demandent la liste de tous les nœuds à utiliser dès le départ. Enfin, la plupart des lanceurs pour réseaux parallèles sont conçus dans l'optique SPMD, c'est-à-dire pour des processus identiques sur tous les nœuds ; ceci est bien évidemment incompatible avec l'approche client/serveur par nature asymétrique de beaucoup d'exécutifs du réparti. Le déploiement d'exécutifs répartis sur des ressources réseaux prévues pour le parallélisme rencontre donc de nombreux obstacles au niveau du modèle d'exécution ; certains impliquent une sévère limitation — durée de vie identique pour les clients et les serveurs et restrictions aux clients sur des nœuds déterminés dès le départ —, alors que la nécessité d'avoir des processus identique implique une modification du modèle d'exécution par exemple en combinant les exécutables des serveurs et de tous les clients potentiels en un seul binaire.

À l'inverse, les réseaux de type répartis ont des mécanismes de déploiement peu adaptés au modèle d'exécution prôné par les exécutifs parallèles. Cependant, il est plus facile de simuler une topologie statique sur une infrastructure qui autorise la dynamicité que l'inverse. La synchronisation n'étant pas assurée par les infrastructures liées au réseau, elle revient dans ce cas à l'adaptateur parallèle/réparti. Enfin, en parallélisme orienté SPMD comme MPI, on suppose en général que tous les nœuds exécutent le même binaire. Ceci peut ne pas être possible avec des ressources du réparti, en particulier à cause d'hétérogénéité du matériel ou du logiciel. Le cas parallèle/réparti reste cependant moins problématique que le cas réparti/parallèle.

Intégration de plusieurs exécutifs. Chaque exécutif apportant son propre modèle d'exécution, l'intégration de plusieurs exécutifs au sein d'un processus n'est pas évidente *a priori* du point de vue du modèle d'exécution. Que se passe-t-il si des exécutifs imposent des choix différents pour la dynamicité, la synchronisation, ou la gestion de processus par groupe ?

Autoriser absolument n'importe quelle combinaison d'exécutifs dans un processus peut effectivement mener à des conflits sur le choix du modèle d'exécution. Cependant, les combinaisons susceptibles d'être utilisées ne devraient pas poser de problème particulier. En effet, les modèles de programmation décrits à la section 3.4 mettent en jeu plusieurs codes parallèles couplés par un exécutif de type réparti ; du point de vue de l'exécutif réparti, chaque code parallèle est considéré comme une seule entité. Les contraintes de synchronisation, de staticité de la topologie ou d'ensemble de nœuds à connaître sont donc encapsulées à l'intérieur d'une entité qui, vue de l'extérieur, se comporte exactement comme un simple nœud du réparti. Les cas les plus pathologiques, par exemple deux codes parallèles déployés sur un ensemble de nœuds pas totalement disjoints (intersection non-vide, ensembles non-confondus), provoquent un conflit sur la synchronisation et l'aspect SPMD, mais ce genre d'utilisation semble marginal au regard des modèles de programmation visés.

La principale problématique au niveau de l'intégration des modèles d'exécution semble être le cas de l'utilisation d'un exécutif de type réparti sur des ressources spécialisées pour le parallélisme, en particulier le client-serveur sur des ressources prévues pour du SPMD. Le cas inverse — parallèle/réparti — demande de fournir des services supplémentaires (gestion de la topologie, synchronisation) mais ne semble pas poser de conflit particulier. Enfin, accorder les modèles d'exécution lors de l'utilisation simultanée de plusieurs exécutifs semble aisé dans les cas courants : c'est-à-dire dans le cas où des entités parallèles sont encapsulées dans des objets ou composants reliés par des mécanismes du réparti.

5.3.2 Adaptation des modèles d'exécution

Il ne paraît pas réaliste de se restreindre aux cas qui satisfont les contraintes sur les modèles d'exécution imposées par un exécutif ou une ressource réseau. Lors d'un déploiement par exemple d'un client et d'un serveur CORBA sur un réseau aux infrastructures prévues pour une exécution SPMD, il est inacceptable de devoir lancer client et serveur en même temps. Il nous semble donc pertinent de proposer des mécanismes qui permettent un modèle client-serveur dynamique au-dessus de ressources accessibles au travers de mécanismes de type SPMD statique.

Modules chargeables. Nous proposons d'organiser notre architecture de plate-forme de communication à l'aide de composants que nous appelons *modules*. Les modules sont des entités à part entière, interagissant avec d'autres modules par l'intermédiaire d'interface déterminées. Ainsi, toute application, code, bibliothèque, exécutif est considérée comme un module. Un module est constitué de code exécutable (code objet dynamique, *bytecode*, scripts, etc.) et d'informations qui le décrivent : type de code, paramètres de configuration, indication de dépendance par rapport à d'autres modules. Les modules sont, par nature, des entités chargeables et déchargeables dynamiquement.

Pour satisfaire les contraintes de lancement, de synchronisation, ou d'uniformité d'exécutable liées à certains réseaux, nous proposons de ne lancer initialement qu'une partie de la plate-forme de communication. Cette partie, appelée *micro-noyau* ne contient que le strict minimum pour initialiser correctement les ressources en fonction de leurs caractéristiques spécifiques. Ainsi, par exemple la contrainte SPMD de certains réseaux est remplie puisque le même *micro-noyau* est lancé sur tous les nœuds. Après initialisation, le micro-noyau est capable de charger d'autres modules pour différencier les nœuds — par exemple charger un serveur CORBA sur un nœud et un client sur un autre nœud. Le fait de considérer les exécutifs eux-mêmes comme des modules, et pas uniquement les applications, permet de charger une combinaison d'exécutifs différente sur chaque nœud si besoin.

Commande à distance. Le *micro-noyau* et les modules chargeables fournissent des mécanismes de base pour un lancement et des combinaisons flexibles. Il faut ensuite utiliser ces briques de base pour recomposer les modèles d'exécution pour chaque type d'exécutif. Les exécutifs et applications de type réparti sont habituellement exécutés sur des ressources qui considèrent les nœuds individuellement, sans synchronisation ni ensemble d'autre nœud connu, avec des chargements et déchargements dynamiques. Ce mode d'exécution est similaire et parfaitement compatible avec le modèle de modules chargeables proposé. De ce fait, même lorsque les ressources sont orientées vers le parallélisme et imposent un modèle d'exécution particulier avec des contraintes SPMD ou de synchronisation, l'ensemble machine + micro-noyau se comporte comme une ressource de type réparti, autorisant la dynamicité, la différenciation des nœuds, et la connexion dynamique sur un réseau déjà initialisé.

Cependant, pour parvenir à ce résultat il est nécessaire de charger les modules des applications et des exécutifs sur le micro-noyau de la plate-forme de communication, c'est-à-dire à l'intérieur d'un processus (au sens Unix du terme) déjà démarré. Puisque c'est le micro-noyau qui réalise les diverses opérations sur les modules, il est nécessaire de pouvoir lui envoyer des commandes après son démarrage de façon à réaliser les opérations dynamiquement sans connaissance particulière avant le démarrage. Il est donc indispensable de mettre en place une procédure de commande à distance des micro-noyaux.

Le modèle d'exécution le plus répandu en parallélisme, à savoir SPMD, peut être également appliqué à l'aide de ce mécanisme. Pour cela, il est nécessaire de charger un module sur un ensemble de nœuds, de lancer l'exécution de façon synchronisée, puis de décharger le module quand tous les nœuds ont terminé l'exécution. Ces opérations peuvent par exemple être réalisées automatiquement par un script qui pilote à distance un ensemble de micro-noyaux. Nous remarquons qu'en plus de permettre des mécanismes d'exécution de type réparti au-dessus de ressources parallèles, le modèle basé sur un micro-noyau et des modules chargeables permet également une plus grande liberté de déploiement des applications parallèles. En effet, dans la mise en œuvre du modèle SPMD décrite ci-dessus, grâce à la séparation des mécanismes spécifique au parallélisme des ressources parallèles, rien n'empêche de déployer deux applications parallèles sur un ensemble disjoints de nœuds d'une seule machine parallèle, ou de déployer une application parallèle sur un ensemble de nœuds à cheval sur deux ou plusieurs machines parallèles, voire plusieurs sites. Les mécanismes de chargements dynamiques synchronisés fournissent la partie relative au modèle d'exécution, le modèle d'abstraction et de virtualisation proposé dans les sections précédentes prend en charge les communications en elles-mêmes.

5.4 Conclusion

Nous avons présenté dans ce chapitre une architecture pour une plate-forme de communication pour les grilles. Son modèle d'abstraction permet de s'affranchir de la frontière entre les différents paradigmes. Cette plate-forme permet d'utiliser des exécutifs existants sur des types de réseaux pour lesquels ils ne sont pas prévus, et permet également de faire cohabiter des exécutifs qui ne sont pas conçus pour. L'architecture proposée est conçue pour s'insérer dans les infrastructures logicielles existantes en étendant leurs possibilités.

Le cœur de cette architecture est constitué d'interfaces abstraites qui servent de point de convergence entre les différents réseaux et les différents exécutifs. Devant la diversité des moyens de communication considérés, nous avons opté pour une différenciation en fonction des paradigmes, et une unification à l'intérieur d'un paradigme. La différenciation permet une adaptation sans compromis superflus d'un exécutif d'un paradigme sur un réseau d'un autre paradigme. Contraindre tous les moyens de communication vers une unique abstraction impose en effet des compromis ou implique de renoncer à des propriétés importantes. L'unification à l'intérieur d'un paradigme permet une bonne flexibilité. À partir du moment où un adaptateur existe vers une abstraction, tous les exécutifs du paradigme peuvent profiter de la méthode ou du réseau pris en charge par cet adaptateur. La couche d'adaptation d'abstraction est conçue pour permettre l'optimisation des cas fréquents en fournissant directement un adaptateur qui le traite. Toute liberté est laissé pour gérer de façon particulière les cas sensibles. Ils ne sont pas contraint artificiellement à emprunter un chemin donné : un adaptateur est libre de fournir n'importe qu'elle interface sur n'importe quel réseau.

Cette architecture est organisée, dès le départ, de façon modulaire et extensible. Il est possible d'intégrer facilement de nouveaux types de réseaux ou de nouvelles méthodes de communication sans remettre en question l'architecture globale. Comme nous le verrons au chapitre 10, cette ouverture des méthodes de communication ne pénalise toutefois pas les performances générales : la plupart du temps, une nouvelle méthode de communication se traduit par le remplacement d'un adaptateur par un autre. La liberté permise par l'utilisation de modules chargeables permet également une grande flexibilité au niveau des modèles d'exécution.

Le modèle proposé peut donc être vu comme un concentrateur (*hub*) de communication. Il rassemble de nombreux exécutifs, réseaux et méthodes de communication, et permet toutes les combinaisons d'assemblage. La liberté laissée aux interfaces — plusieurs interfaces selon plusieurs paradigmes — permet des adaptations efficaces. Les puissants mécanismes d'adaptation d'abstraction et de virtualisation par personnalité font que c'est la plate-forme de communication qui s'adapte à son environnement plutôt que l'inverse.

Le modèle de gestion des communications est organisé selon les trois niveaux : accès arbitré, adaptation des abstractions, virtualisation des interfaces. Les trois chapitres suivants détaillent chacun de ces niveaux en commençant par le cœur— l'adaptation d'abstraction —, puis la virtualisation d'interface, et enfin l'accès arbitré aux ressources.

Chapitre 6

Des abstractions de communication multi-paradigme

Sommaire

6.1 Principe de la couche d'adaptation d'abstraction 75
6.1.1 Paradigmes et niveaux 76
6.1.2 Propriétés des abstractions 76
6.1.3 Notion d'adaptateur 77
6.2 L'abstraction du réparti 78
6.2.1 Caractéristiques 78
6.2.2 Interface abstraite pour le réparti 79
6.2.3 Adaptateurs pour l'abstraction du réparti 81
6.3 L'abstraction du parallélisme à mémoire distribuée 82
6.3.1 Caractéristiques 82
6.3.2 Interface abstraite pour le parallélisme à mémoire distribuée 83
6.3.3 Adaptateurs pour l'abstraction du parallélisme à mémoire distribuée 84
6.4 L'abstraction du parallélisme à mémoire partagée 85
6.5 Généralisation des mécanismes d'adaptation d'abstraction 86
6.5.1 Variétés d'adaptateurs 86
6.5.2 Composition et combinaisons d'adaptateurs 87
6.5.3 Exemples de variantes d'adaptateurs 89
6.6 Sélection automatique et assemblage des adaptateurs 90
6.7 Conclusion 92

Ce chapitre présente la partie centrale de l'architecture proposée : l'adaptation d'abstraction. Ce niveau est le cœur d'une plate-forme multi-paradigme qui supporte des exécutifs et des réseaux basés sur plusieurs paradigmes. C'est à ce niveau que la traduction d'un paradigme à l'autre est réalisée.

La présentation de ce niveau commence par une description générale des principes de l'adaptation d'abstraction, puis nous l'illustrons au travers des exemples de trois abstractions : pour le calcul réparti, pour le parallélisme à mémoire distribuée, et pour le parallélisme à mémoire partagée. Nous décrivons ensuite les mécanismes généralisés des adaptateurs en tant que composants interchangeables, puis les mécanismes de sélection de la méthode de communication appropriée. Enfin, nous concluons par une discussion sur la complexité et l'adéquation de cette approche par rapport aux besoins.

6.1 Principe de la couche d'adaptation d'abstraction

Dans cette section, nous décrivons les principes fondamentaux de la couche d'adaptation d'abstraction : les paradigmes, les propriétés des abstractions, et la notion d'adaptateur.

6.1.1 Paradigmes et niveaux

La couche d'adaptation d'abstraction est la partie de notre architecture de plate-forme de communications qui permet le passage d'un paradigme à un autre. Elle utilise l'interface offerte par l'accès arbitré, selon le paradigme de la ressource considérée, et propose au-dessus une interface générique du même paradigme ou d'un autre paradigme.

Les interfaces offertes par la couche d'adaptation d'abstraction sont de niveau abstrait, c'est-à-dire qu'elles sont totalement indépendantes des ressources utilisées. De plus, ces interfaces sont génériques : à l'intérieur d'un paradigme, l'interface de programmation est unifiée en une unique API qui permet d'exprimer toutes les propriétés nécessaires pour réaliser des communications suivant le paradigme considéré. Nous appelons une telle interface générique de niveau abstrait une *abstraction*. La couche d'adaptation d'abstraction présente donc autant d'abstractions que de paradigmes voulus au niveau générique.

Les interfaces utilisées par la couche d'adaptation d'abstraction sont de niveau arbitré, c'est-à-dire qu'elles sont dépendantes des ressources utilisées ; leur paradigme est spécifique selon les ressources, et éventuellement l'API elle-même peut dépendre de la méthode d'accès. Cependant, ces interfaces d'accès aux ressources de communications sont arbitrées : plusieurs clients peuvent l'utiliser en même temps sans problèmes de réentrance ni risque d'iniquité, y compris en cas de stratégies d'utilisation différentes.

Au niveau de l'accès arbitré — niveau bas, par rapport à cette partie —, le paradigme sur lequel l'interface est conçue est décidé par la nature de la ressource. Au niveau de l'abstraction — niveau haut de cette partie —, le paradigme sur lequel est basé l'interface générique est décidé par la nature de la personnalité, donc par le paradigme de l'exécutif. La couche d'adaptation d'abstraction a donc pour but de fournir une interface générique de niveau abstrait au-dessus de la ressource disponible, quels que soient les paradigmes respectifs de l'API abstraite voulue et de la ressource de communication réellement utilisée. À chaque niveau, les paradigmes considérés sont :

- le calcul réparti tel que défini par la définition 2.17 ;
- le parallélisme à mémoire distribué tel que défini par la définition 2.15 ;
- le parallélisme à mémoire partagée tel que défini par la définition 2.16.

À chaque paradigme correspond une abstraction.

6.1.2 Propriétés des abstractions

Les différentes abstractions manipulent des notions communes du calcul concurrent. Nous définissons les différentes abstractions par l'intermédiaire des notions suivantes :

Adressage — Les mécanismes d'adressage sont les mécanismes qui permettent de nommer les nœuds de calcul par l'intermédiaire d'un numéro logique ou d'une adresse absolue ; l'adressage peut également être implicite dans certains cas.

Topologie — La description de la topologie est une description des nœuds connus et de leurs interconnexions.

Établissement de connexion — L'établissement de connexion consiste en la construction de la topologie. Il peut être dynamique ou statique, c'est-à-dire modifiable ou non en cours de session.

Échange de messages — Les communications sont réalisées au travers de primitives d'échange de messages. Nous distinguons les primitives d'envoi et de réception.

Clôture de connexion — La clôture de connexion consiste en l'arrêt de l'échange de messages, et la destruction de la topologie.

Nous définissons les propriétés des trois abstractions correspondant aux trois paradigmes :

Calcul réparti — En réparti, l'adressage se fait par une adresse absolue, c'est-à-dire que l'adresse désigne une ressource de façon unique à l'échelle mondiale. Le standard le plus répandu est le système IP. La topologie connue de chaque nœud est constituée d'un seul lien point-à-point à la fois : chaque nœud "connaît" le nœud à l'autre extrémité du lien ; les différents liens sont indépendants les uns des autres. L'établissement et la clôture de connexion sont dynamiques : ils

peuvent intervenir à tout moment en cours de session. Le modèle d'établissement de connexion est le client/serveur (les primitives `connect/accept`). L'échange de messages est basé sur la notion de *flux* : les frontières des messages peuvent ne pas être les mêmes du point de vue du récepteur et de l'émetteur. La synchronisation entre l'émetteur et le récepteur est relâchée : un envoi peut terminer avant que le récepteur ne soit prêt à recevoir.

Parallélisme à mémoire distribuée — En parallélisme à mémoire distribuée, l'adressage se fait par numérotation logique des nœuds à l'intérieur d'un *monde* connu. L'adressage est alors immédiat, sans nécessiter de résolution d'adresse. La topologie connue de chaque nœud est alors le *monde*, constitué d'un ensemble défini de nœuds. La topologie est statique, c'est-à-dire que la création et destruction du *monde* sont des opérations synchrones, réalisées en même temps par tous les nœuds. L'envoi et la réception de messages sont optimisées pour les performances (latence et débit de transfert). La réception par messages actifs permet une bonne latence en contexte concurrent ; l'envoi/réception est basé sur l'empaquetage/dépaquetage incrémental pour permettre des opérations zéro copie même avec des messages complexes de longueur variable. En plus des échanges de messages point à point, l'abstraction comprend des opérations collectives de type diffusion/réduction *scatter/gather*), et des primitives de synchronisation (barrière).

Parallélisme à mémoire partagée — En parallélisme à mémoire partagée, l'adressage des nœuds est implicite : c'est au système — logiciel ou matériel — que revient de localiser l'emplacement physique des données et de s'assurer de leur disponibilité sur le nœud qui en a besoin. La topologie est, comme en parallélisme à mémoire distribuée, basée sur la notion de *monde*. L'envoi et la réception de messages sont des opérations qui n'ont pas de sens pour ce paradigme puisque l'accès aux données se fait par accès à la mémoire ; les transferts sont implicites, pris en charge par le système.

6.1.3 Notion d'adaptateur

La couche d'adaptation d'abstraction est entièrement modulaire, construite à l'aide de composants assemblables dynamiquement par une tierce partie. Chaque composant fournit une abstraction de communication, c'est-à-dire une interface générique de niveau abstrait. Nous appelons un tel composant d'adaptation d'abstraction un *adaptateur*. Un adaptateur utilise les services de la couche d'accès arbitré pour fournir une abstraction selon le paradigme choisi. Le niveau d'adaptation d'abstraction de la plate-forme de communications est donc constitué d'une collection d'adaptateurs, chaque adaptateur fournissant une abstraction sur une ressource.

Un adaptateur est donc un composant dont le but est d'offrir une interface abstraite au-dessus des différentes ressources de communication. Une couche d'adaptation d'abstraction complète — permettant n'importe quelle abstraction sur n'importe quelle ressource — est donc constituée d'une collection d'adaptateurs réalisant toutes les combinaisons entre les abstractions et les ressources. Nous appelons une telle collection une *collection principale* d'adaptateurs. Une collection principale d'adaptateurs pour trois paradigmes est schématisée à la figure 6.1.

Les ressources qu'un adaptateur utilise en fonction de l'abstraction fournie définissent alors trois catégories d'adaptateurs.

Adaptateurs *directs* — il s'agit d'adaptateurs qui offrent une abstraction du même paradigme que la ressource qu'ils utilisent. Il s'agit seulement d'une adaptation des services bas niveau fournis par les méthodes d'accès en l'abstraction adéquate. C'est par exemple un adaptateur qui implémente une abstraction de type réparti sur un réseau de type réparti. Dans certains cas, l'implémentation des adaptateurs directs est triviale.

Adaptateurs *croisés* — il s'agit d'adaptateurs qui offrent une abstraction d'un autre paradigme que la ressource qu'ils utilisent, en effectuant une traduction *trans-paradigme*. C'est par exemple un adaptateur qui implémente une abstraction parallèle sur un réseau de type réparti ou inversement. La complexité des adaptateurs croisés est variable, selon les propriétés respectives des paradigmes fournis et utilisés.

Adaptateurs *généralisés* — les adaptateurs généralisés sont des méthodes de communication supplémentaires, en dehors de la collection principale. Ils peuvent être alternatifs aux adaptateurs

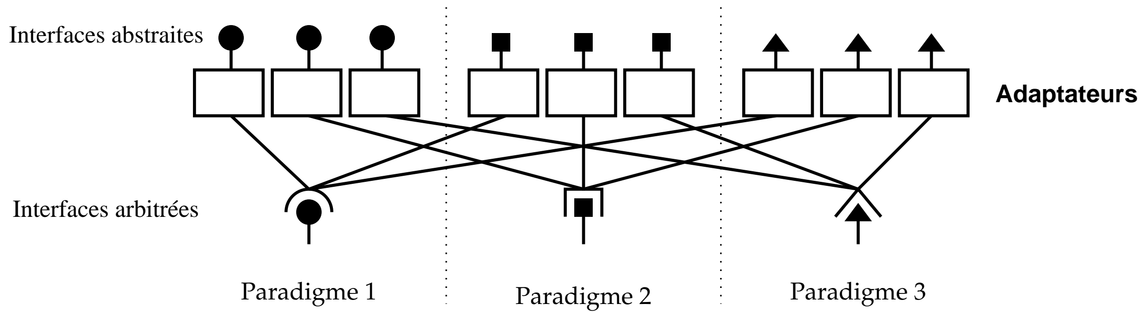

FIG. 6.1 – *Représentation d'une collection principale d'adaptateurs pour trois paradigmes — chaque symbole représente un paradigme.*

directs et croisés, proposer des méthodes de communication supplémentaires, ou encore s'intercaler dans des assemblages. Nous détaillerons les mécanismes d'adaptateurs généralisés à la section 6.5.

Une collection principale d'adaptateurs fournissant chacune des trois abstractions sur des ressources selon trois paradigmes contient donc neuf adaptateurs, que l'on nommera par le couple constitué de l'abstraction fournie et du paradigme de la ressource utilisée. Ce nombre se décompose en trois adaptateurs directs, et six adaptateurs croisés, seuls certains adaptateurs croisés étant d'une complexité notable.

6.2 L'abstraction du réparti

Dans cette section, nous présentons une abstraction pour les communications de type "réparti" (au sens de la définition 2.17), de notre plate-forme de communications, ainsi que ses diverses incarnations sous forme d'adaptateurs qui présentent comme interface l'abstraction du réparti.

6.2.1 Caractéristiques

Une interface abstraite pour le réparti est destinée à être utilisée par une personnalité qui fournit une API de type réparti, telle que *socket* BSD, Posix AIO ou X/Open XTI. Elle doit donc être suffisamment générique pour recouvrir les notions exprimables par ces différentes interfaces. Une abstraction du réparti est caractérisée par :

Topologie — La topologie en réparti est basée sur la notion de *lien*. Un *lien* est une connexion logique, bidirectionnelle, construite dynamiquement, reliant deux nœuds. La construction d'un lien est asymétrique, avec la notion de client (actif) et de serveur (passif, attend les clients) ; après sa construction, un lien a un comportement symétrique. Les liens sont indépendants les uns des autres : les opérations de connexion, déconnexion, et échange de messages sur les différents liens n'interfèrent pas. Un lien est manipulé par son *descripteur*, une structure de donnée qui décrit son état courant et qui permet d'y faire référence.

Adressage — De par les caractéristiques d'hétérogénéité des réseaux de type réparti, plusieurs systèmes d'adresses coexistent. Il ne semble donc pas légitime de privilégier un système d'adressage plutôt qu'un autre. Une abstraction du réparti exprime les adresses selon le système d'adressage de la ressource considérée. C'est aux adaptateurs que revient d'interpréter les adresses qu'ils sont capables de comprendre en fonction des ressources qu'ils gèrent.

Asynchronisme — les opérations d'échange de messages et d'établissement de connexion existent sous forme synchrone (bloquante) — *ie.* l'appel de fonction retourne quand l'opération est terminée — et sous forme asynchrone (non-bloquante) — *ie.* l'appel de fonction retourne immédiatement, sans attendre la terminaison de l'opération. En réparti, il est courant de gérer plusieurs opérations simultanément à l'aide des versions non-bloquantes.

Il est relativement aisé de dériver une version synchrone de primitives asynchrones ; il suffit d'ajouter des mécanismes de synchronisation. Le contraire est plus complexe — cela nécessite de créer des *threads* qui travaillent en arrière plan —, moins efficace, et consomme plus de ressources. Pour pouvoir supporter des personnalités à API synchrones et asynchrones, et les couches inférieures de la plate-forme étant elles-mêmes basées sur des mécanismes asynchrones pour un meilleur support de la concurrence, notre abstraction du réparti est donc basée sur des mécanismes asynchrones.

Protocole — Il existe plusieurs types de protocoles en réparti ; en nomenclature "*sockets*", ils sont désignés par exemple par les constantes `SOCK_STREAM` et `SOCK_DGRAM`. L'API ne change pas d'un protocole à l'autre mais la sémantique des primitives varie — fiabilité, frontières de messages, etc. sont gérés différemment selon le protocole. Nous intégrons les différents protocoles dans notre modèle d'abstraction en les considérant comme des ressources différentes, donc gérées par un adaptateur différent. Par exemple, ce sont des adaptateurs différents qui fournissent TCP ou UDP. Cependant, ces adaptateurs partagent la majeure partie de leur code ; c'est pourquoi dans la suite nous ne distinguerons pas la plupart du temps les différents protocoles.

6.2.2 Interface abstraite pour le réparti

Les adaptateurs qui réalisent l'abstraction du réparti sont des composants qui présentent cette interface abstraite. Cette interface abstraite est subdivisée en trois parties : gestion de lien (connexion et déconnexion), échange de données, et scrutation et contrôle de la synchronisation. L'interface manipule essentiellement trois type d'objets : des *adresses*, des *liens* (par l'intermédiaire de leur *descripteur*), et des *observateurs* (décrits ci-dessous). Nous décrivons d'abord le principe général de l'interface abstraite du réparti puis nous présentons successivement chacune des trois parties de l'interface.

Principe général de l'interface abstraite du réparti. Notre modèle d'interface abstraite pour le réparti présente la caractéristique d'être asynchrone. Les opérations sont démarrées par un appel de fonction puis continuent en tâche de fond, principalement par le biais des mécanismes asynchrones des niveaux inférieurs. Pour consulter l'état d'avancement et détecter la terminaison des opérations, deux méthodes principales existent : la scrutation et l'interruption. La scrutation consiste en la consultation de l'état d'avancement ; l'interruption consiste en l'invocation par le composant qui a réalisé l'opération d'une fonction de rappel quand celle-ci est terminée. Il est difficile de dissocier les deux : en effet, la scrutation seule résulte en des attentes actives quand on souhaite une synchronisation explicite, ce qui nuit à la concurrence ; l'interruption seule ne permet pas de consulter l'état d'avancement d'une opération avant sa terminaison. Notre abstraction générique pour le réparti propose ces deux modes de contrôle de l'asynchronisme.

Scrutation et contrôle d'état d'avancement. Le contrôle de la synchronisation et de la terminaison des opérations est confié à une entité que nous appelons *observateur*. Un observateur est un objet qui réside à l'intérieur d'un adaptateur et dont une instance est attaché à chaque descripteur de lien. Les interactions entre l'utilisateur et l'observateur sont assurées par une interface dédiée de l'adaptateur. Un observateur permet à la fois la scrutation de l'état d'un lien, de déclencher des interruptions à la terminaison des opérations, et d'attendre explicitement la terminaison en attente passive. À chaque descripteur de lien est associé un observateur, appelé *observateur primaire*. Il permet de gérer la synchronisation des opérations réalisées sur son lien associé. Nous appelons *observateur secondaire* un observateur qui n'est pas associé à un lien mais à d'autres observateurs.

La scrutation à l'aide d'un observateur est effectuée à l'aide d'*attributs* ; un attribut est un champ d'information d'un descripteur pour une opération particulière. Les attributs sont organisés en trois catégories :

– attributs d'état global : état du descripteur (ouvert, peut écrire, peut lire, etc.), notification d'erreur ;

- attributs de gestion de lien : adresse locale, adresse du pair distant, connexions entrantes en attente ;
- attributs de gestion de transferts : boîte aux lettres d'émission, boîte aux lettres de réception.

Les primitives de manipulation d'attributs offertes par un adaptateur sont la *consultation* d'un attribut, et éventuellement sa *modification*.

La synchronisation à l'aide d'un observateur est effectuée par le biais d'évènements appelés *signaux*. L'adaptateur qui gère l'observateur génère un signal pour notifier des évènements particuliers comme la terminaison normale d'une opération ou la détection d'une erreur. Un signal est accompagné de l'étiquette de l'attribut auquel il se rapporte. Les primitives de manipulation des signaux des observateurs offertes par un adaptateur sont l'*attente* passive d'un signal, et l'enregistrement d'une *fonction de rappel* qui sera appelée par l'observateur quand il recevra un signal correspondant à l'attribut spécifié. Ainsi, les deux modes de synchronisation par scrutation et interruption sont possibles. Il est alors possible de réaliser une synchronisation explicite de trois façons :

- attente sur l'observateur, puis contrôle de l'état par consultation de l'attribut associé (mode dit "attente passive") ;
- enregistrement dans l'observateur d'une fonction de rappel au démarrage de l'opération, et invocation de cette fonction par l'observateur quand l'opération signale sa terminaison (mode dit "interruption") ;
- création d'un observateur secondaire enregistré dans les observateurs de plusieurs descripteurs, puis synchronisation sur cet observateur secondaire à l'aide de l'une des deux méthodes précédentes (mode dit "agrégé" : attente agrégée ou interruption agrégée).

L'association de primitives asynchrones et du mécanisme d'observateur ouvre la voie à des utilisations variées de l'abstraction du réparti. Les propriétés de l'observateur permettent des implémentations efficaces tout en pouvant profiter de la composabilité des adaptateurs. En effet, grâce aux attributs, la lourdeur d'une encapsulation systématique peut être évitée — la plupart des implémentations de consultation des attributs peuvent se résumer à prendre un verrou et accéder directement aux champs publics du descripteur —, et la composition est assurée par le passage systématique par les services de l'adaptateur pour la *consultation* et la *modification* des attributs. De plus, les observateurs secondaires, associés non pas à un lien mais à d'autres observateurs, permettent des observations agrégées, briques de base d'opération telles que `select()` ou `poll()`.

Opérations de gestion de lien. L'interface abstraite pour le réparti comprend quatre primitives pour la gestion des liens :

lier — cette primitive a pour but de lier un descripteur de lien à une adresse physique. L'adresse est donnée en format dépendant du protocole voulu (typiquement, TCP/IP). L'adaptateur l'interprète en fonction des possibilités d'adaptation qu'il offre. Il traduit l'adresse de niveau abstrait en une adresse physique, en faisant au besoin le travail d'adaptation. Un même adaptateur peut comprendre plusieurs types d'adresses. Cette primitive est synchrone : elle termine immédiatement sans bloquer.

écouter — cette primitive fait d'un lien existant un serveur en attente de connexion en provenance du réseau. À partir de l'appel à cette primitive, les demandes de connexions entrantes sont traitées. Une connexion entrante résulte en la création d'un nouveau lien, sans affecter le serveur ; les connexions fraîchement établies sont mises en attente dans la file de connexions du descripteur. Cette primitive est le pendant de la primitive *sockets* `listen` ; cependant, l'abstraction ne contient pas de primitive équivalente à `accept` des *sockets* BSD : les nouvelles connexions sont directement récupérées de façon asynchrone dans la file des connexions en attente *via* l'observateur. Pour chaque connexion entrante établie, un signal est envoyé à l'observateur du descripteur du lien serveur.

connecter — cette primitive établit un lien en tant que client, vers un serveur dont l'adresse est donnée en paramètre. L'adresse obéit aux mêmes contraintes que pour la primitive *lier*. Le serveur doit être en mode d'écoute de son côté pour que la connexion puisse se faire. Cette primitive est asyn-

chrone, elle génère un signal à l'observateur du descripteur quand la connexion est réellement établie.

fermer — cette primitive clôture un lien dans un sens ou l'autre ou les deux. Après l'invocation de cette primitive, il n'est plus valide d'invoquer une réception ou une émission. Cette primitive est asynchrone : il n'est pas obligatoire que la confirmation de clôture soit immédiate ; elle est alors notifiée par un signal à l'observateur. C'est seulement après clôture bidirectionnelle et confirmation que le descripteur peut effectivement être détruit.

Ce sont ces primitives qui réalisent la dynamicité de la topologie propre au réparti.

Opérations de transfert de données. Les transferts de données sont basés sur des flux continus (*streams*) ou des datagrammes en fonction du protocole. Ils sont réalisés au travers des deux primitives *envoyer* et *recevoir*. Celles-ci initialisent l'envoi ou la réception de données ; l'état d'avancement est consulté dans les *boîte d'émission* et *boîte de réception via* l'observateur. Une opération peut ainsi être à zéro, terminée, mais aussi dans un état intermédiaire dans lequel seulement une partie des données a été traitée. L'adaptateur est libre de générer des signaux à l'observateur pour ces états intermédiaires quand il juge que l'évènement mérite d'être signalé, et obligatoirement quand l'intégralité des données a été traitée.

Il est possible d'interrompre une opération en cours avant sa terminaison complète. Il suffit pour cela de remettre à zéro la boîte d'émission ou de réception. Dans certain cas, la boîte est marquée "non-interruptible" ce qui indique que la remise à zéro n'est pas possible immédiatement (par exemple si l'adaptateur sous-jacent utilise un mécanisme de rendez-vous) ; dans ce cas, il suffit d'attendre le prochain signal de l'observateur pour effectuer la remise à zéro. Cette fonctionnalité est particulièrement utile pour la personnalité des *sockets* BSD, qui s'arrêtent fréquemment sur une opération partiellement terminée ; par exemple, un `read` peut retourner avant la réception totale de la taille de donnée demandée.

L'interface abstraite que doit présenter un adaptateur pour l'abstraction du réparti est constituée d'un ensemble de telles primitives :

- fabrique : *constructeur*, *destructeur* de liens ;
- observateur : *consulter*, *modifier* des attributs, *attendre* un signal, *enregistrer* une fonction de rappel ;
- gestion de lien : *lier*, *écouter*, *connecter*, *fermer* ;
- transfert de données : *envoyer*, *recevoir*.

Une implémentation de ce modèle d'interface abstraite pour le réparti est décrite en section 9.3.1.

6.2.3 Adaptateurs pour l'abstraction du réparti

Nous décrivons ici les trois adaptateurs de base qui offrent l'interface abstraite du réparti présentée ci-dessus.

Adaptateur réparti direct. L'adaptateur réparti direct a pour but d'offrir l'interface abstraite du réparti au-dessus des ressources de communications de type réparti. Il utilise les services de la couche d'accès arbitré pour accéder à ces ressources. L'idée principale de tout adaptateur direct est la transparence maximum ; un adaptateur direct doit conserver sur toute la chaîne d'implémentation les propriétés importantes du paradigme de la ressource utilisée. Dans le cas du réparti, il s'agit de conserver la dynamicité, la topologie gérée avec la granularité du lien, l'interopérabilité, et les flux continus (*streaming*).

Il est primordial que l'adaptateur direct ne réalise que l'adaptation "triviale" et ne réalise pas d'adaptation supplémentaire. Ceci est nécessaire pour que les exécutifs qui utilisent l'adaptateur réparti direct soient interopérables "sur le fil" (*ie.* au niveau des messages TCP/IP) avec des exécutifs qui utilisent directement TCP/IP. Mise à part cette contrainte d'interopérabilité, l'adaptateur réparti direct est l'un des plus simples.

Adaptateur croisé : réparti/parallèle à mémoire distribuée. L'adaptateur réparti sur parallèle à mémoire distribué a pour but d'offrir l'interface abstraite du réparti au-dessus des réseaux haute performance utilisés en parallélisme. Il doit offrir l'illusion d'un réseau de type réparti en utilisant un réseau de type parallèle au travers de la couche d'accès arbitré. Les adaptations majeures ont lieu au niveau de la gestion de lien et des transferts de données. Pour la gestion de lien, il s'agit d'offrir des connexions et déconnexions logiques dynamiques gérées lien par lien, alors que le réseau offre une clique statique. Cet adaptateur gère donc un multiplexage en utilisant comme clef de multiplexage le numéro de port TCP ou UDP. Les mécanismes de transfert de données doivent être adaptés pour tenir compte des spécifications du protocole fourni : flux continus ou datagrammes, stockage dans des tampons ou suppression des paquets inattendus.

Adaptateur croisé : réparti/parallèle à mémoire partagée. L'adaptateur réparti sur parallèle à mémoire partagée offre l'interface abstraite du réparti au-dessus d'une zone de mémoire partagée — en accès direct dans le même processus, ou par un segment de mémoire partagé de type IPC System V. Cet adaptateur fonctionne d'une façon très similaire à l'adaptateur réparti sur parallèle à mémoire distribuée sauf que les messages sont postés dans une boîte aux lettres hébergée par le segment partagé au lieu d'être envoyés sur le réseau. Cet adaptateur peut en quelque sorte être considéré comme un pilote *loopback* qui assure également les communications d'un nœud avec lui-même.

6.3 L'abstraction du parallélisme à mémoire distribuée

Dans cette section, nous décrivons l'abstraction de communication de type parallèle à mémoire distribuée, telle que posée par la définition 2.15, de notre plate-forme de communications. Dans un premier, temps nous présentons ses caractéristiques, puis nous décrivons les adaptateurs qui présentent une telle interface.

6.3.1 Caractéristiques

Au niveau abstrait en parallélisme, aucun standard [1] ne se dégage réellement. Nous proposons donc une abstraction aussi générique que possible, qui pourra facilement être utilisée par diverses personnalités en fonction des besoins.

Cette abstraction est basée sur une topologie logique de clique, que nous appelons *circuit*. Un circuit est construit sur un groupe de nœuds, ensemble ordonné, sans doublons, numéroté, et invariant dans le temps. Un circuit peut être créé à tout moment en cours de session, mais à partir du moment où un circuit est créé, son contenu ne peut pas être modifié ; il est toutefois possible de le détruire et d'en créer de nouveaux à tout moment. Il est envisageable d'étendre ce modèle vers une dynamicité "restreinte", *ie.* l'ajout et la suppression de nœuds étant vus comme des reconfigurations. Cette contrainte topologique a pour contrepartie la facilité d'adressage : du point de vue de l'abstraction, l'adresse d'un nœud est simplement son numéro d'ordre dans le circuit — l'adressage est immédiat. Chaque nœud peut communiquer avec tous les autres de son circuit. La construction et la destruction d'un circuit sont des opérations synchrones : l'opération sur chacun des nœuds ne retourne que lorsque tous les autres nœuds sont prêts et ont également terminé l'opération. Cette approche est donc particulièrement adaptée au modèle de programmation SPMD. Un circuit est une entité logique de communication. Un nœud peut prendre part à plusieurs circuits, les différents circuit pouvant être basés sur des groupes de nœuds différents.

Notion de *sous-circuit*. Cette abstraction a pour particularité de pouvoir mettre en jeu plusieurs adaptateurs pour un seul circuit — pourtant une seule entité de communication du point de vue logique. En effet, si un circuit est à cheval sur deux machines parallèles reliées par un réseau de type réparti, une communication peut emprunter un réseau parallèle ou un réseau réparti selon la paire de

1. MPI est un standard destiné à être utilisé par des applications et non des exécutifs. Ce n'est pas une interface générique de niveau abstrait.

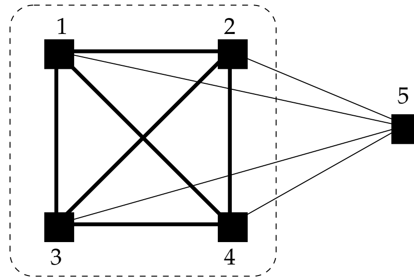

FIG. 6.2 – *Exemple de décomposition en sous-circuits : un sous-circuit basé sur l'adaptateur direct regroupe les nœuds 1 à 4 reliés par un réseau rapide ; le nœud 5, en dehors de la grappe, est relié à chacun des nœuds 1 à 4 par un sous-circuit parallèle/réparti distinct.*

nœuds considérée. Nous découpons les circuits en entités appelées *sous-circuits*. Chaque nœud peut apparaître dans plusieurs sous-circuits, en fonction de ses connexions. La contrainte de construction des sous-circuits est que chaque couple de nœuds du circuit doit apparaître dans exactement un sous-circuit, et chaque sous-circuit est géré par un seul adaptateur — tous les nœuds à l'intérieur d'un sous-circuit peuvent communiquer avec tous les autres nœuds du même sous-circuit à l'aide du même adaptateur.

La façon exacte de découper un circuit en sous-circuits est décidée par le sélecteur (décrit en section 6.6) en fonction de ses connaissances de la topologie. Les solutions privilégiées sont celles qui maximisent le nombre de nœuds inclus dans des sous-circuits basés sur l'adaptateur direct. La figure 6.2 présente l'exemple d'un circuit qui recouvre cinq nœuds : les quatre premiers nœuds sont sur une même grappe disposant d'un réseau rapide ; ils sont donc inclus dans un sous-circuit basé sur l'adaptateur direct (parallèle/parallèle). Le nœud 5 est en dehors ; quatre sous-circuits de type croisé (adaptateur parallèle/réparti) sont donc construits pour encapsuler avec une abstraction parallèle les liens 1–5, 2–5, 3–5 et 4–5. Le circuit qui relie les cinq nœud est l'union de ces cinq sous-circuits.

Chaque sous-circuit est lui-même un circuit, avec l'hypothèse supplémentaire qu'il n'utilise qu'un seul adaptateur. Chaque sous-circuit est donc géré par l'adaptateur adéquat. Le circuit qui regroupe les sous-circuits, appelé *circuit composite*, est géré quant à lui par un adaptateur dédié à l'assemblage des sous-circuits, à l'aiguillage des requêtes vers le ou les adaptateurs mis en jeu dans l'opération demandée, et à la traduction des numérotations des nœuds. Dans les cas simples, un composite peut ne regrouper qu'un seul sous-circuit composé de tous les nœuds du circuit.

Cette approche permet de gérer efficacement les sous-circuits comme des entités constituées de plusieurs nœuds, et donc utiliser leurs opérations collectives le cas échéant. La décomposition permet d'encapsuler les spécificités de chaque réseau dans les adaptateurs, tout en permettant à un circuit de franchir la frontière entre les réseaux. Ainsi, les adaptateurs de type parallèle/parallèles expriment dans leurs sous-circuits la capacité de gérer les ensembles de nœuds. Les adaptateurs de type parallèle/réparti encapsulent un lien (deux nœuds) dans un sous-circuit. Notre approche de la décomposition qui demande que chaque paire de nœuds puisse être connectée par un adaptateur simplifie le circuit composite — il n'a pas besoin de gérer de routage, simplement une décision de lien au départ — et sépare les méthodes avancées d'interconnexion avec routage (tunnels, par exemple) du concept de circuit, car ce sont deux notions orthogonales. Ceci rend ces méthodes d'interconnexion utilisables avec d'autres abstractions. Une approche "diagonale" (*ie.* qui combine les problématiques d'interconnexion et d'abstraction de type parallèle) disperse les efforts sans apporter de réel gain ni en performance ni en simplicité dans ce cas.

6.3.2 Interface abstraite pour le parallélisme à mémoire distribuée

L'interface abstraite pour le parallélisme à mémoire distribuée est scindée en trois catégories de primitives : l'administration du circuit, l'échange de messages, et les opérations collectives.

Administration du circuit. L'administration d'un circuit est réalisée par un *constructeur* et un *destructeur* de circuit. La construction se base sur un groupe de nœuds donné par l'utilisateur. La création est ensuite directement réalisée par l'adaptateur, ou déléguée à des adaptateurs secondaires en suivant les instructions du sélecteur, suivant que le circuit est un sous-circuit ou un composite. Lors de la construction du circuit, l'utilisateur désigne les nœuds directement par leur adresse dans le système d'adressage propre de la plate-forme, ce qui impose de ne créer un circuit que sur des nœuds déjà connus de la plate-forme ; ceci n'est pas une contrainte dans la mesure où il n'est pas prévu de construire des circuits dont tous les nœuds n'utiliseraient pas le même plate-forme de communication.

Échanges de messages. L'échange de message sur un circuit est basé sur une interface très largement inspirée de celle de Madeleine 2, réputée être suffisamment expressive tout en permettant une implémentation particulièrement efficace. Ses caractéristiques principales sont une construction incrémentale des messages, et les contraintes de transfert données au niveau sémantique plutôt que fonctionnel. Cependant, à l'inverse de Madeleine, nous basons la réception sur des messages actifs plutôt qu'une réception explicite. Ceci facilite le démultiplexage et la concurrence, et est utilisé par les méthodes d'accès aux ressources. Ce principe est propagé dans l'abstraction du parallélisme. C'est à l'exécutif que revient de réaliser des réceptions explicites ou non. Les échanges sont organisés en *messages*, chaque message étant composé d'un ou plusieurs paquets. Chaque envoi est accompagné de la méthode à employer. Ces contraintes sont décrites au niveau sémantique, contrairement à la plupart des autres environnements qui reposent sur une description au niveau fonctionnel.

Opérations collectives. Les opérations collectives de l'interface abstraite sont des briques de base pour des opérations plus complexes. Elles n'ont pas vocation à réaliser un jeu d'opérations collectives complet, mais uniquement de pouvoir exprimer les opérations susceptibles d'être optimisées par les niveaux inférieurs du fait de leur connaissance de la topologie ou de capacités de certains matériels à réaliser des opérations collectives. Ces primitives expriment à un niveau générique ces propriétés. Ces opérations collectives de base sont :

barrière — il s'agit d'une barrière de synchronisation : tous les nœuds s'attendent mutuellement.

diffusion — un nœud envoie une information à tous les autres nœuds.

L'interface abstraite que doit présenter un adaptateur pour l'abstraction du parallélisme à mémoire distribuée est constituée d'un ensemble de telles primitives :

– administration : *constructeur* et *destructeur* de circuit ;
– messages : primitives incrémentales à la façon Madeleine ;
– opérations collectives : *barrière* et *diffusion*.

Un implémentation de ce modèle d'interface abstraite pour le parallélisme à mémoire distribuée est décrite en section 9.3.2.

6.3.3 Adaptateurs pour l'abstraction du parallélisme à mémoire distribuée

Nous décrivons ici les adaptateurs de base utilisés pour offrir l'interface abstraite du parallélisme à mémoire répartie : les trois adaptateurs pour chacun des trois paradigmes, et l'adaptateur composite pour gérer les circuits composites.

Adaptateur direct parallèle à mémoire distribuée. L'adaptateur parallèle à mémoire distribuée direct offre l'interface proposée sur un ensemble de nœuds reliés par un réseau de type parallèle. Cet adaptateur se doit d'être capable d'exploiter au mieux les capacités du matériel qui, dans ce cas, est basé sur le même paradigme que l'interface demandée. Il est donc souhaitable de conserver les propriétés suivante : API d'échange de messages zéro-copie, gestion de l'ensemble des nœuds comme une seul entité sans revenir au lien par lien qui perd la topologie, exploitation des possibilités d'opérations collectives offertes. Il va de soit que lorsque le réseau est géré à bas niveau — au niveau de l'accès arbitré — avec la bibliothèque Madeleine, cet adaptateur est particulièrement trivial !

Adaptateur croisé : parallèle/réparti. Cet adaptateur présente l'interface abstraite du parallélisme à mémoire distribuée au-dessus des réseaux de type réparti. Il sert donc à gérer des sous-circuits de type réparti. De tels sous-circuit abstraient avec un point de vue parallèle des *liens*, entité de base de la topologie du réparti. Les sous-circuits gérés par cet adaptateur ont donc pour particularité d'être composés toujours d'exactement deux nœuds.

L'adaptation des mécanismes de construction de circuit – symétriques en parallélisme — est donc basé sur des mécanisme asymétriques client/serveur — un nœud est client, l'autre est serveur. L'échange de messages et basique ; sur un réseau TCP/IP qui utilise déjà des mécanismes d'agrégation de paquets dans son implémentation dans le système, nul pour l'application de réordonnancer ou agréger les paquets. Chaque `pack/unpack` est directement traduit en `send/recv`, à l'exception des opérations en mode `send_LATER` qui sont différées. Enfin, les opérations collectives sont particulièrement triviales quand la topologie est restreinte à deux nœuds.

Adaptateur croisé : parallèle à mémoire distribuée/partagée. Cet adaptateur présente une interface à passage de message au-dessus d'une ressource à mémoire partagée. L'adaptation dans ce sens est relativement aisée. Deux cas sont considérés, selon que les appels de fonction peuvent être effectués directement d'un nœud à l'autre :

- les nœuds partagent le même processus : la création d'un sous-circuit consiste en une barrière, l'envoi de message consiste en l'invocation directe du traitant de réception par l'émetteur, la barrière est un sémaphore, et les autres opérations collectives sont réalisées de façon directe.
- seul un segment de mémoire est partagé : l'appel direct du traitant n'est pas possible dans ce cas. Un mécanisme de boîte aux lettres est mis en place dans le segment partagé ; l'essentiel des opérations se ramène alors au cas d'une ressource parallèle à mémoire distribuée, en utilisant la boîte aux lettres dans le segment comme mécanisme de passage de messages. Les opérations collectives sont optimisées grâce à la possibilité de lecture/écriture par plusieurs nœuds dans la boîte au lettres.

Adaptateur composite. L'adaptateur composite est le composant qui réalise l'essentiel de l'abstraction de la topologie et permet à un circuit de s'affranchir de la topologie physique. Sauf cas exceptionnels, c'est l'adaptateur composite qui est utilisé par les personnalités et les exécutifs. Les autres adaptateurs ne servent la plupart du temps que pour les *sous-circuits* ; il sont donc utilisés par l'adaptateur composite et pas directement par les exécutifs.

Lors de la création d'un circuit, l'adaptateur composite s'en remet au *sélecteur* pour décider de la topologie du découpage en sous-circuits. L'adaptateur réalise ensuite la création proprement dite, en délégant aux adaptateurs adéquats pour chaque sous-circuit. Puisque l'initialisation est bloquante et dépend des autres nœuds, et puisque chaque nœud peut créer plusieurs sous-circuits, il y a risque d'interblocage. Pour éviter l'interblocage, tous les nœuds réalisent les opérations de construction et destruction de sous-circuit dans le même ordre, selon l'ordre lexicographique de l'identifiant des nœuds qui composent chaque sous-circuit.

Pour les opérations d'échange de messages simples, l'adaptateur composite choisit simplement quel sous-circuit utiliser — un seul choix est possible pour un nœud destination donné — et réalise la translation d'adresse pour se placer dans le référentiel du sous-circuit concerné. Pour les opérations collectives, l'adaptateur composite est chargé d'effectuer l'agrégation qui permet de reconstituer l'opération sur le circuit composite à partir de l'opération effectuée sur chacun des sous-circuits.

6.4 L'abstraction du parallélisme à mémoire partagée

Dans cette section, nous traitons de la place du parallélisme à mémoire partagée dans notre modèle d'abstraction. En parallélisme à mémoire partagée, les interfaces de programmation sont nombreuses : interface `pthread`, API *TreadMarks* [40], API *JiaJia*, déclarations par macros, `put/get` de type *Cray* ; les interfaces sont variées et une application écrite pour l'une de ces interfaces demande un pénible travail d'adaptation pour passer à une autre interface.

Le principe d'une décomposition entre une interface abstraite générique pour le paradigme et diverses personnalités qui incarnent les différentes API est applicable également au paradigme de la mémoire partagée. Cette approche a été notamment mise en œuvre dans HAMSTER [166], la MVP de SMiLE [167]. HAMSTER offre en effet pas moins de neuf interfaces différentes au-dessus de sa propre interface. Cette approche est viable : porter HAMSTER dans notre plate-forme de communication pourrait permettre d'offrir les neuf interfaces supportées.

Cependant, le cas de la mémoire partagée est différent des deux autres paradigmes car, contrairement aux autres, un système qui offre une interface à mémoire partagée sur une architecture à mémoire distribuée est un exécutif en soi. De plus, il ne semble pas raisonnable de choisir une approche qui utiliserait plusieurs adaptateurs pour fournir cette interface sur un même nœud, en fonction du paradigme réparti ou parallèle à utiliser pour joindre les nœuds distants concernés. Une MVP est déjà suffisamment complexe !

C'est pourquoi nous ne considérons pas le parallélisme à mémoire partagée comme une abstraction à fournir par un adaptateur, mais plutôt comme un exécutif. Par soucis de simplicité, un tel exécutif est à construire au-dessus des circuits de l'abstraction parallèle à mémoire distribuée ou de l'une de ses personnalités. L'adaptation de paradigme au-dessus de ressources arbitraires se fait donc en deux temps : une adaptation vers le parallélisme à mémoire distribuée par les mécanismes d'abstraction présentés dans les sections précédentes, puis une adaptation du parallélisme à mémoire distribuée vers le parallélisme à mémoire partagée. Il est alors possible d'utiliser *a priori* n'importe quelle implémentation de MVP pour couvrir tous les cas. Ceci permet de mettre à profit toutes les fonctionnalité d'une MVP, y compris des fonctionnalités non-exprimables dans le système de HAMSTER comme une gestion poussée des protocoles de cohérence.

6.5 Généralisation des mécanismes d'adaptation d'abstraction

Nous présentons ici une généralisation du mécanisme des adaptateurs présenté au début de cette partie sur la couche d'adaptation d'abstraction à la section 6.1.3. Nous présentons tout d'abord cette généralisation basée sur des variantes et l'assemblage d'adaptateurs. Nous présentons ensuite quelques exemples qui tirent profit de cette généralisation.

6.5.1 Variétés d'adaptateurs

Pour toutes les abstractions sur tous les types de ressources, il existe un adaptateur dans la collection principale. Cependant, cet adaptateur n'est pas forcément le meilleur choix en toutes circonstances. D'un point de vue général, le seul but d'un adaptateur est d'offrir une interface abstraite ; pour fournir ces services, il est libre d'utiliser une ressource, plusieurs ressources, un autre adaptateur, ou même une bibliothèque venant d'une tierce partie. Pour gérer ces cas qui ne sont pas satisfaits par une collection principale d'adaptateurs, nous présentons dans les paragraphes suivants des mécanismes qui vont au-delà des possibilités d'une collection principale. Nous y présentons trois types d'adaptateurs : les adaptateurs alternatifs, les intercepteurs, et les adaptateurs croisés abstraits.

Alternatives. Un *adaptateur alternatif* est un adaptateur qui réalise une adaptation alternative à un adaptateur d'une collection principale : il utilise et offre les mêmes interfaces qu'un adaptateur d'une collection principale, et peut être utilisé à sa place. Par exemple, dans le cas d'une adaptation réparti/réparti, la collection principale contient un adaptateur direct qui ne réalise aucune transformation sur les accès. L'utilisateur peut cependant vouloir une autre méthode de communication dans certains cas : par exemple une compression à la volée ou des flux parallèles pour obtenir de meilleures performances, ou une authentification pour une meilleure sécurité. Ce sont autant de moyens différents d'offrir une interface abstraite de type réparti au-dessus d'un réseau de type réparti. En ce sens, des adaptateurs basés sur ces différentes méthodes sont des *alternatives* à l'adaptateur direct réparti/réparti. La figure 6.3 illustre trois alternatives possibles pour l'adaptation réparti/réparti. Ces trois alternatives sont utilisables de façon interchangeable.

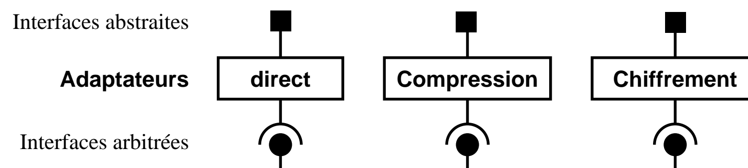

FIG. 6.3 – *Exemple de trois alternatives d'adaptateurs réparti/réparti : à gauche, l'adaptateur direct ; au milieu et à droite, deux alternatives utilisables à sa place — les différents symboles indiquent les* niveaux *d'interface.*

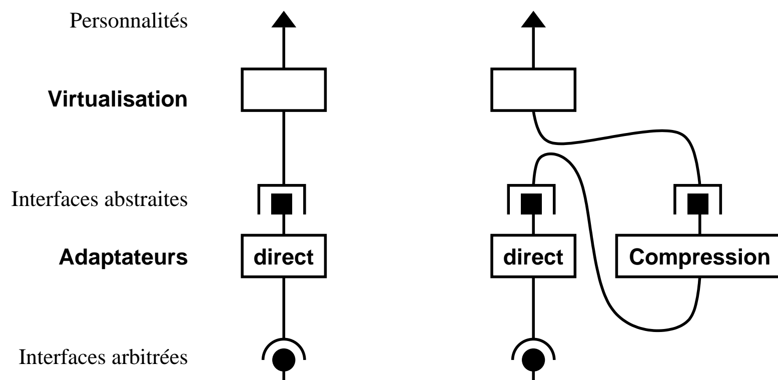

FIG. 6.4 – *Exemple d'intercepteur : à gauche, une personnalité utilise l'adaptateur direct ; à droite, l'intercepteur "compression" s'interpose entre la personnalité et l'adaptateur direct — les différents symboles représentent les* niveaux *d'interface.*

Intercepteurs. Un *intercepteur* est un adaptateur qui utilise une interface abstraite de même type que celle qu'il fournit. Il peut ainsi être inséré dans une chaîne d'utilisation de composants pour ajouter une propriété. La figure 6.4 illustre l'exemple d'un intercepteur réparti/réparti — il fournit et utilise une interface abstraite de type réparti — qui compresse les données à la volée. Un intercepteur peut utiliser n'importe quel adaptateur, pas nécessairement un adaptateur direct.

Adaptateurs croisés abstraits. Un adaptateur *croisé abstrait* est un adaptateur qui présente une interface abstraite au-dessus d'une interface abstraite d'un autre paradigme. Il est donc *croisé* car il réalise une traduction trans-paradigme ; il est abstrait dans le sens ou il ne travaille qu'avec des interfaces abstraites. La figure 6.5 représente une collection composée de deux adaptateurs directs et deux adaptateurs croisés abstraits (deux paradigmes seulement sont représentés pour raison de clarté).

Le principal attrait des adaptateurs croisés abstraits réside dans les perspectives de compositions qu'ils ouvrent. Ils rendent possible l'utilisation des adaptateurs alternatifs et des intercepteurs depuis n'importe quel paradigme. Il est ainsi par exemple possible d'utiliser la compression sur les parties d'un circuit qui empruntent un réseau lent, sans développer une version spécifique de l'adaptateur de compression pour le parallélisme.

6.5.2 Composition et combinaisons d'adaptateurs

Un adaptateur peut utiliser un ou plusieurs adaptateurs, pouvant eux-mêmes utiliser un ou plusieurs adaptateurs. Il s'agit alors de *composition*. L'éventail des combinaisons est donc large.

Décomposition et recomposition. Un assemblage d'un adaptateur croisé abstrait avec un adaptateur direct introduit un doublon avec un adaptateur croisé d'une collection principale. En effet, comme illustré par la figure 6.6, la présence d'un adaptateur croisé abstrait autorise deux façons différentes d'offrir une interface abstraite du paradigme 2 au-dessus d'une ressource réseau programmée selon le

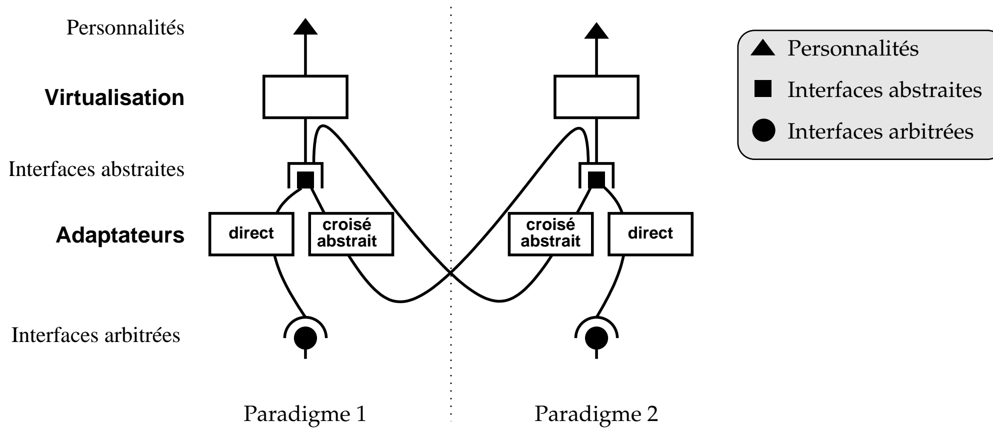

FIG. 6.5 – *Deux adaptateurs croisés abstraits au-dessus de deux adaptateurs directs — les différents symboles représentent les* niveaux *d'interface.*

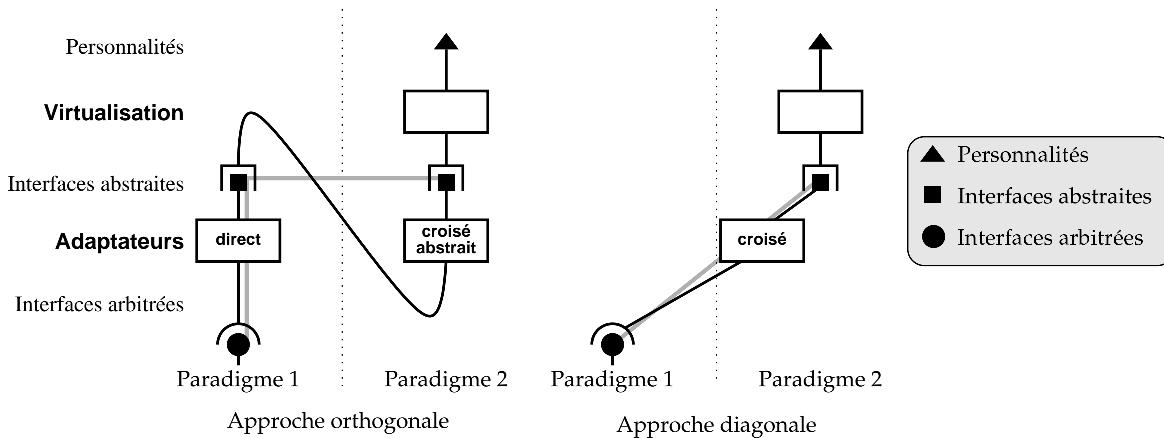

FIG. 6.6 – *Adaptateur croisé abstrait et adaptateur croisé : à gauche, assemblage d'un adaptateur croisé abstrait et d'un adaptateur direct (décomposition orthogonale) ; à droite, adaptateur croisé seul (approche diagonale).*

paradigme 1. À gauche, l'adaptation est découpée en deux parties *orthogonales* (orientations représentées en grisé) : l'adaptateur direct réalise une adaptation *verticale* — interface de plus haut niveau, sans changer de paradigme ; l'adaptateur croisé abstrait réalise une adaptation *horizontale* — interface de même niveau d'abstraction, avec changement de paradigme. À droite, l'adaptateur croisé réalise une adaptation *diagonale* : il offre une interface de plus haut niveau d'abstraction *et* d'un autre paradigme que celle qu'il utilise. L'adaptateur croisé peut être vu comme un raccourci équivalent à l'assemblage adaptateur croisé abstrait + direct. En termes de fonctionnalités, il est équivalent à l'assemblage. Dans certains cas, l'adaptateur qui adopte une approche diagonale permet d'éviter des compromis superflus qui sont induits par le passage par une interface abstraite d'un autre paradigme ; dans d'autres cas, l'approche diagonale n'apporte rien par rapport à une décomposition orthogonale.

Par exemple, un adaptateur parallèle à mémoire distribuée sur réparti fait doublon avec l'assemblage composé de l'adaptateur croisé abstrait parallèle à mémoire distribuée sur réparti et l'adaptateur direct réparti/réparti ; cependant, l'interposition de l'abstraction du réparti (basée sur la réception explicite) entre l'interface abstraite du parallélisme à mémoire distribuée (basée sur des messages actifs) et de l'interface arbitrée du réparti (basée également sur des messages actifs) pénalise les performances. Dans cet exemple, l'adaptateur croisé principal — basé sur une approche diagonale — semble donc tout de même pertinent.

À l'inverse, il ne semble pas très pertinent de proposer deux adaptateurs qui présentent chacun

une interface abstraite à mémoire partagée, l'un sur des ressources répartie, l'autre sur des ressources parallèles distribuées. La principale difficulté réside en effet dans l'émulation d'une mémoire partagée au-dessus d'une mémoire distribuée, soit-elle orientée vers le parallélisme ou le réparti. Il est donc plus judicieux dans ce cas de ne proposer qu'un seul adaptateur croisé pour l'interface abstraite parallèle à mémoire partagée : au-dessus de l'interface abstraite à mémoire distribuée.

Le choix des adaptateurs à implémenter, en évitant les assemblages qui font doublons ou non, dépend principalement du rapport entre le gain de performance ou de facilité d'utilisation et de l'effort de développement nécessaire.

Alternatives ou intercepteurs : choix du niveau d'implémentation. Les adaptateurs alternatifs et les intercepteurs sont très proches dans leurs buts. Nous remarquons qu'il est par exemple envisageable d'implémenter une compression à la volée aussi bien sous la forme d'un adaptateur alternatif que d'un intercepteur. Rares sont les cas où l'on voudra implémenter les deux. La décision d'implémenter l'un plutôt que l'autre est prise en fonction de la nature de chaque méthode de communication. Une méthode intimement liée à un type de réseau particulier, requérant un contrôle fin, est implémentée sous forme d'adaptateur alternatif. Une méthode générale, pouvant être utilisée au-dessus de différents adaptateurs (l'adaptateur direct et les alternatives), susceptible d'être composée avec d'autres méthodes, est implémentée sous forme d'intercepteur.

Par exemple une communication par flux parallèles est très liée aux particularités de TCP/IP ; étant donné qu'elle demande une gestion fine de l'entrelacement des envois, il est souhaitable de l'implémenter aussi bas que possible. De plus, elle n'a pas grand sens sur d'autres types de réseaux, ou du moins pas avec les mêmes réglages. Cette méthode est donc fournie sous forme d'adaptateur alternatif, utilisant directement la couche d'accès arbitré au réseau de type réparti.

À l'inverse, une méthode de communication qui compresse les données à la volée est considérée, d'un point de vue logique, comme un *filtre* qui s'interpose sur un flux de données. Le principe de la compression n'est en rien lié à la nature du protocole utilisé à bas niveau. Dans ce cas, il est judicieux d'offrir la compression sous forme d'intercepteur.

Discussion. Adaptateur alternatifs, intercepteurs et adaptateurs croisés abstraits ajoutent des méthodes de communication à une collection principale d'adaptateurs. Le fait de baser l'adaptation d'abstraction sur un assemblage de composants rend cette couche intrinsèquement évolutive et ouverte : il suffit de remplacer un composant par un autre pour offrir une nouvelle façon d'utiliser les réseaux. De ce fait, il n'y a pas besoin de mécanisme de *plug-ins* : les méthodes de communication ajoutées *a posteriori* utilisent les mêmes infrastructures que celles prévues au départ. Ou, autrement dit, tous les adaptateurs sont des *plug-ins* dès le départ. Bien que décrites à part des méthodes "principales" dans ce document pour des raisons de clarté, les méthodes dites "alternatives" sont des adaptateurs à part entière, au même titre que les méthodes principales.

6.5.3 Exemples de variantes d'adaptateurs

Nous décrivons ici quelques exemples de variantes d'adaptateurs en dehors de la collection principale. Les exemples présentés sont tous basés sur l'abstraction du réparti. Ceci n'est guère surprenant dans la mesure où en parallélisme, les exécutifs utilisent une API et des services de relativement bas niveau pour accéder au réseau. Cependant, ces adaptateurs peuvent être utilisés par des exécutifs du parallélisme pour leurs liens qui empruntent un réseau de type réparti. La liste données n'est pas exhaustive ; elle n'est donnée que dans le but d'illustrer les possibilités offertes par l'architecture de la plate-forme proposée. Les exemples présentés sont, pour la plupart, spécifiques aux problématiques des WAN et des pare-feux.

Compression à la volée — Sur des réseaux dont le débit est faible par rapport à la puissance de calcul des processeurs des nœuds, il est rentable de compresser les données à la volée au moment de les envoyer et de les décompresser en réception, tel que mis en œuvre dans AdOC [114] par exemple. Ceci est intéressant quand le facteur limitant est la bande passante disponible sur un

lien, par exemple un modem ou un lien trans-continental lent. La compression à la volée étant *a priori* insérable sur tout type de lien, il est intéressant de la fournir sous la forme d'un intercepteur capable d'être inséré sur n'importe quel adaptateur fournissant une abstraction du réparti.

Flux parallèles — Dans le cas de réseaux longue distance (WAN) à latence élevé et haut débit, la moindre perte sur le réseau est catastrophique pour TCP/IP ; la latence élevée fait que la perte est détectée tardivement, ce qui dégrade les performances, d'autant plus qu'avec le haut débit la moindre interruption se traduit en une baisse significative du débit effectif. Par expérience, il a été constaté [39] qu'utiliser plusieurs flux TCP/IP en parallèle pour un flux logique améliore globalement les performances dans ce cas. Ceci amortit les pertes en les isolant à un seul flux ; de plus, l'adaptation aux conditions de trafic (par exemple le *slow start*) est plus rapide. Ces mécanismes étant optionnels, à régler au cas par cas, il est pertinent de les fournir dans un adaptateur spécifique utilisable de façon transparente par les exécutifs. Étant donné la très forte dépendance vis-à-vis du bas niveau (taille de fenêtre TCP, nombre de flux, et facteur d'entrelacement étant liés), ces mécanismes s'inscrive dans notre architecture sous la forme d'un adaptateur alternatif.

Connexion passive — Quand deux sites désirent établir des liens, et que l'un des deux sites est isolé par un pare-feu (*firewall*) qui bloque toutes les connexions entrantes, toutes les connexions doivent être établies dans le même sens, initiées par le site protégé. En disposant d'un canal qui transporte des messages de contrôle (lui-même établi à l'initiative du site protégé), nous construisons un adaptateur qui simule des connexions normales. Quand une demande de connexion dans le sens interdit est demandée par l'application, l'adaptateur envoie une requête par le canal de contrôle pour que, à bas niveau, la connexion soit établie dans le sens passant.

Authentification — L'authentification vise à établir que les deux parties établissant une connexion sont bien ceux qu'ils prétendent être. Les méthodes utilisées sont basées sur des techniques cryptographiques, souvent suivant le standard est TLS [68] (*Transport Layer Security*), variante ouverte de `Ssl` qui est propriétaire.

Chiffrement — Le chiffrement vise à empêcher qu'une tierce partie ne puisse lire un message échangé. Là encore, les mécanismes utilisés sont issus de techniques cryptographique. Le chiffrement est prévu dans TLS [68] en plus de l'authentification.

Tunnel — L'adaptateur "tunnel" permet de faire passer une connexion de bout-en-bout sur une chaîne constituée de plusieurs tronçons, chacun pouvant être géré par un adaptateur différent. L'adaptateur "tunnel" se charge de faire suivre les données d'un tronçon à l'autre. Par soucis de généricité, cet adaptateur fournit une abstraction du réparti et utilise une abstraction du réparti pour chaque tronçon. Un tunnel peut par exemple être utilisé pour fournir une connectivité de type réparti avec l'extérieur à des nœuds d'une machine parallèle dont tous les nœuds ne sont pas accessibles par IP.

Protocole à tolérance de pertes — Sur les réseaux longue distance IP qui associent un taux de perte élevé et une forte latence, les mécanismes de récupération mis en œuvre par TCP peuvent s'avérer particulièrement inefficaces. Parfois, TCP passe énormément de temps à retransmettre une donnée, à telle point que pour l'application, le temps que la donnée arrive, elle est "périmée". Dans ce cas, il peut être souhaitable d'utiliser un protocole à tolérance de perte pour gagner en performance en renonçant à ré-émettre des données perdues quand on préfère des données partielles à jour plutôt que des données complètes. Un tel compromis entre la fiabilité et la performance est réalisé par exemple par le protocole VRP [14], un protocole avec une tolérance de perte ajustable. Les paramètres pour contrôler la tolérance de perte et les retours d'informations sur les pertes effectives étant des informations nécessaires à manipuler mais spécifique à ce protocole, il fournit sa propre interface de programmation différente de l'abstraction du réparti standard de notre plate-forme.

6.6 Sélection automatique et assemblage des adaptateurs

Dans cette section, nous présentons les bases des mécanismes de sélection automatique et d'assemblage des adaptateurs. La sélection et l'assemblage des adaptateurs sont régis par une entité appelée

sélecteur. Le but du sélecteur est de trouver un assemblage d'adaptateurs qui offre l'*abstraction* voulue pour établir une connexion donnée.

Description des ressources. Les choix du sélecteur sont guidés par une connaissance de la *topologie* du réseau, une connaissance des *capacités* des adaptateurs, et de *préférences* données par l'utilisateur. La *topologie* est dynamique, constituée d'information sur les machines, les réseaux, et les connexions réseaux. La description de la topologie est constituée des entités suivantes :

hôtes — un hôte est une machine, au sens habituel du terme. Une machine à plusieurs processeurs régie par une seule instance d'un système d'exploitation est un seul hôte.

nœuds — un nœud est un processus géré par la plate-forme de communication, où *processus* s'entend au sens Unix du terme. À chaque nœud correspond une instance de la plate-forme de communication. Un nœud est hébergé par un hôte ; plusieurs nœuds peuvent partager le même hôte.

réseaux — un réseau est une ressource qui relie plusieurs nœuds (deux ou plus). Sur un *réseau*, tous les nœuds peuvent contacter directement tous les autres nœuds.

point de connexion — un point de connexion est une référence à un descripteur de connexion d'un adaptateur. Du point de vue du sélecteur, ce sont seulement des références vers des objets opaques, manipulables par l'adaptateur qui a enregistré la référence dans le sélecteur. Un point de connexion est lié à son paradigme.

La *topologie* statique du monde connu est constituée des hôtes, et des réseaux connus, et de leurs relations. Pour pouvoir prendre en compte les hôtes *internet* en dehors du monde connu de la plate-forme, nous introduisons un réseau spécial, appelé "*internet*". Par défaut, quand il ne connaît pas de descripteur de nœud pour l'adresse demandée, le sélecteur suppose que le nœud est un hôte connecté à ce réseau *internet* par défaut. La topologie dynamique est constituée des nœuds et des points de connexion. Les *capacités* des adaptateurs sont décrites à l'aide des notions suivantes :

interface — une interface (API) peut être l'une des abstractions, une interface spécifique de la couche d'accès arbitré, ou encore une interface spécifique à une méthode (par exemple VRP).

méthode — une méthode de communication identifie le protocole de communication mis en œuvre par un adaptateur. La méthode est "directe" lorsqu'elle ne réalise aucune transformation. Ceci permet en particulier de savoir si l'interopérabilité est assurée. Une méthode peut être par exemple "compression", "chiffrement", etc. décrite indépendamment de l'adaptateur qui l'implémente et du paradigme.

Le sélecteur connaît une collection d'adaptateur. Chaque adaptateur de la collection est décrit par la ou les interfaces qu'il fournit et qu'il utilise, ainsi que les éventuelles méthodes de communication qu'il met en œuvre. Chaque *réseau* de la topologie est caractérisé par l'*interface* à employer pour l'emprunter.

Règles et préférences. Le sélecteur effectue la sélection automatique en se basant sur un ensemble de règles. Une règle est constituée d'un filtre pour décider quand appliquer la règle, et de la méthode à utiliser dans ce cas. C'est une approche comparable aux filtres mis en œuvre dans la plupart des logiciels de courrier électronique : chaque règle contient une partie pour détecter quand appliquer la règle, et une partie décrivant l'action à réaliser lors de l'application.

Les préférences sont des règles fixées par l'utilisateur pour guider le sélecteur dans ses choix. Les préférences peuvent être données comme paramètres de configurations, et peuvent également être changées en cours d'exécution. En l'absence de préférences particulières, le sélecteur utilise des règles par défaut qui consistent en des choix raisonnables : n'utiliser aucune méthode de transformation, *ie.* n'utiliser que des adaptateurs de la collection principale.

Chaque demande de décision d'assemblage est accompagnée d'informations pour décrire la connexion à établir : réseau ou nœud, abstraction voulue. Les filtres des règles sont alors appliqués successivement jusqu'à trouver une règle dont le filtre correspond ; la règle donne alors une méthode et/ou un réseau à utiliser. Ceci permet d'exprimer des préférences comme par exemple : "utiliser la compression pour joindre tel ensemble de machines", "utiliser toujours l'authentification sur TCP/IP".

Sélection d'assemblage. Le sélecteur cherche un *assemblage* qui offre le service demandé sur les ressources connues. Les requêtes au sélecteur sont constituée de :
- l'*interface* demandée — habituellement une abstraction ;
- la spécification des *ressources* visées : réseau, nœud, groupe de nœuds ou point de connexion sur un autre nœud ;
- de façon optionnelle, une chaîne de *méthodes* de communication.

La sélection se fait alors en deux temps :
1. choix d'une *chaîne de méthodes* de communications : si une méthode de communication est donnée lors de la requête, elle outrepasse les préférences ; dans le cas contraire, les règles des préférences sont parcourues et donnent la chaîne de méthodes à utiliser.
2. *résolution* de la chaîne de méthodes *en assemblage* d'adaptateurs : à partir des propriétés des adaptateurs et de la chaîne de méthodes, le sélecteur détermine les adaptateurs à utiliser.

La réponse de du sélecteur est alors un point de connexion construit pour l'interface demandée. Le point de connexion contient en particulier les informations d'assemblage des adaptateurs de la connexion qu'il décrit.

Ces mécanismes de sélection automatiques conviennent pour des utilisations simples, quand les choix sont guidés par les *possibilités* ; cependant, ces mécanismes sont trop rudimentaires et pourraient être améliorés pour tous les cas où plusieurs choix sont possibles et que le choix est alors guidé par une adaptation aux conditions. Idéalement, ce n'est pas à l'utilisateur de décider quelle méthode de communication utiliser, mais plutôt à la plate-forme de choisir en fonction de sa connaissance des propriétés des réseaux et des méthodes de communication. L'étude de ces mécanismes d'adaptation est un prolongement logique de notre proposition.

6.7 Conclusion

Nous avons présenté dans cette section le niveau qui réalise l'adaptation d'abstraction. La caractéristique essentielle de ce niveau est qu'il est capable d'offrir n'importe laquelle des interfaces abstraites sur n'importe laquelle des ressources prises en charge par la plate-forme, indépendamment de leur paradigme respectif. L'adaptation d'abstraction est effectuée par des composants appelés *adaptateurs*, chaque adaptateur prenant en charge la présentation d'un type d'interface sur un type de ressource. Des assemblages d'adaptateurs permettent de composer les méthodes de communication.

Ce modèle est un compromis entre la multiplication du nombre d'adaptateurs et l'efficacité de l'implémentation. Le nombre des abstractions étant peu variable, et la diversité des ressources prises en charge étant relativement caché par la couche d'accès arbitré, le contenu de la collection principale d'adaptateurs est peu variable. De plus, peu d'adaptateurs sont complexes. Il n'est pas nécessaire d'implémenter la totalité des adaptateurs croisés ; dans le cas où l'approche orthogonale convient (adaptateur direct + adaptateur croisé abstrait), l'adaptateur croisé ne présente aucun intérêt. Il s'agit alors de déterminer et d'optimiser les cas dignes d'intérêt.

La couche d'adaptation d'abstraction fournit un moyen homogène et cohérent d'accéder à toutes les ressources réseaux prises en charge par la plate-forme de communication. Ces accès ont lieu au travers de l'une des abstractions proposées, le paradigme de l'abstraction étant laissé libre à l'utilisateur. L'utilisation de ces interfaces abstraites par des exécutifs se fait par l'intermédiaire de personnalité pour présenter diverses interfaces au-dessus des abstractions. Les mécanismes de personnalités font l'objet du chapitre suivant.

Chapitre 7

Virtualisation des interfaces de communication

Sommaire

7.1 Contexte d'intégration 93
7.2 Principe de la virtualisation des ressources 94
7.3 Virtualisation par personnalités 96
7.3.1 "Portabilité" d'une abstraction sous n exécutifs 96
7.3.2 Quelle API générique? 97
7.3.3 Notion de personnalité 97
7.4 Transparence 98
7.4.1 Transparence d'utilisation 98
7.4.2 Transparence de protocole 100
7.5 Discussion sur la virtualisation 101
7.6 Conclusion 101

Dans ce chapitre, nous étudions la virtualisation des interfaces de communication comme moyen pour apporter *portabilité* et *passerelles trans-paradigme* de façon transparente à des exécutifs. Nous définissons la *virtualisation* comme suit :

Définition 7.1 : ***virtualisation*** — La *virtualisation* consiste en l'émulation d'une ressource (dite *virtuelle*) sur une autre ressource (dite *réelle*). Les exécutifs utilisent la ressource virtuelle comme s'ils utilisaient une ressource réelle ; la ressource virtuelle simule le comportement de la ressource réelle. Ceci permet, sans les modifier, de faire utiliser par des exécutifs des ressources pour lesquelles ils ne sont pas conçus.

Dans la suite de cette section, les figures représentent les empilement de couches logicielles. Les boîtes symbolisent des composants. Par convention, nous utilisons une police de caractère différente pour distinguer les implémentations des composants et leurs *interfaces*. Cette section commence par une discussion sur la pertinence d'une réutilisation des codes existants puis décrit et positionne les mécanismes de virtualisation dans le cadre d'une plate-forme de communications pour les grilles.

7.1 Contexte d'intégration

Nous étudions ici la position d'une plate-forme de communications pour les grilles de calcul par rapport aux autres infrastructures logicielles utilisées pour programmer les grilles. Une telle plate-forme de communication n'est pas un logiciel isolé ; il est primordial qu'elle s'intègre dans l'architecture globale. L'« esprit de la grille » est, comme nous l'avons vu au chapitre 3, d'utiliser et d'agréger

des ressources de calcul existantes pour réaliser un ensemble plus puissant. Nous verrons que ce principe est valable également au niveau logiciel : les grilles de calcul sont programmées principalement en utilisant un assemblage d'infrastructures logicielles existantes agrégées.

Les codes de calculs, existants ou conçus spécifiquement pour les grilles, sont écrits par des spécialistes ; chacun de ces codes demande une grande expertise dans un domaine particulier différent pour chaque code. Ils sont donc écrits nécessairement par des personnes différentes. Or chaque équipe, chaque développeur, ou chaque domaine d'expertise, a ses propres besoins et habitudes de développement logiciel. Le choix du modèle de programmation, du langage, ou encore de l'exécutif est donc du ressort du développeur de l'application et n'est pas dicté par une plate-forme particulière. Une plate-forme pour une grille de calcul doit donc nécessairement pouvoir intégrer ces codes variés.

Une profusion d'exécutifs. Les applications mises en œuvre sur les grilles de calculs étant susceptibles d'être basées sur des modèles de programmation variés, avec des exécutifs variés, une plate-forme de communications pour de telles applications se doit de supporter tous les exécutifs demandés. Or, les exécutifs à supporter sont nombreux. D'une part, il sont variés en types : MPI, CORBA, HLA, SOAP, PVM, JVM, et de nouveaux types d'exécutifs à venir. D'autre part, pour chaque type d'exécutif il existe plusieurs implémentations, chaque implémentation ayant ses spécificités ; par exemple pour CORBA, il existe OmniORB, MICO, TAO, ORBacus, etc. ; pour MPI, il existe MPICH, LAM MPI, MPI /Pro, etc. Il est courant que les différentes implémentations des exécutifs aient chacune des fonctionnalités spécifiques en marge des standards ; certaines applications existantes que l'on voudrait déployer sur une grille utilisent ces fonctionnalités spécifiques. Il est donc préférable de supporter un large éventail d'exécutifs, éventuellement plusieurs implémentations différentes d'un même type d'exécutif.

Le développement d'un exécutif est une tâche particulièrement lourde. À titre d'exemple, MICO contient 80 000 lignes de code C++, OmniORB 60 000 lignes de C++, la JVM Kaffe plus de 100 000 lignes de C, ou encore MPICH plus de 200 000 lignes de C. De plus, la conception d'un exécutif requiert des compétences spécifiques, en particulier une bonne maîtrise des normes (MPI, SOAP ou CORBA, par exemple). Ces normes sont en évolution constante ; par exemple, la norme CORBA vient de passer des 2.x à 3.0, MPI est passé de la version 1.0 à 2.0, *Sun* fait évoluer régulièrement la norme Java, et SOAP est en cours d'évolution. Il en résulte que les implémentations des exécutifs sont en perpétuelle évolution, sans toujours garder une compatibilité avec les anciennes versions. À titre d'exemple, d'octobre 2000 à mai 2003, sept nouvelles versions de MICO ont vu le jour, dont certaines comprenant des développements logiciels très lourds (composants CORBA, par exemple).

Il est donc non seulement intéressant de supporter plusieurs types d'exécutifs, mais aussi plusieurs implémentations d'un type d'exécutif, et même plusieurs versions d'une même implémentation. Il semble également préférable de pouvoir prendre en compte facilement et rapidement des exécutifs à venir. Il ne nous semble pas raisonnable de redévelopper tous les exécutifs spécialement pour une plate-forme particulière de communications. La section 4.2.3 a en effet montré que les plates-formes basées sur la reconstruction d'exécutifs souffrent de nombreux défauts, en particulier la difficulté d'intégrer de nombreux exécutifs différents à jour. C'est ainsi par exemple que *Quarterware*, plate-forme générique de construction d'exécutifs, se limite à des sous-ensembles des normes MPI et RMI Java. Puisque nous souhaitons que ces implémentations suivent les évolutions régulières des normes et comprennent les particularités des différentes implémentations, nous choisissons de réutiliser directement les implémentations existantes, de façon à tirer profit des efforts de développement faits par les développeurs de chaque exécutif experts dans leur domaine.

7.2 Principe de la virtualisation des ressources

Nous étudions ici les méthodes pour utiliser des exécutifs existants sur des ressources réseaux pour lesquelles ils ne sont pas prévus. Pour éviter de devoir redévelopper de nombreux exécutifs, comme l'a montré la section précédente, il est en effet primordial de pouvoir utiliser ceux qui existent sur les ressources variées des grilles.

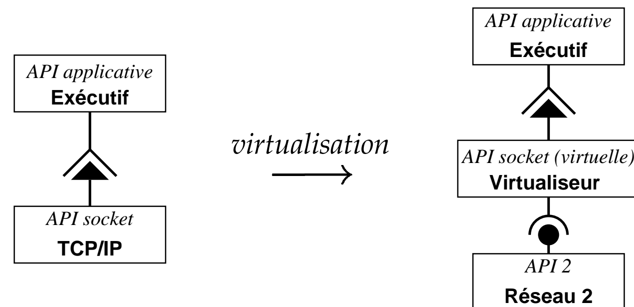

FIG. 7.1 – *Principe de la virtualisation : l'exécutif conçu pour utiliser l'interface ▲ utilise un réseau basé sur l'interface ● grâce au virtualiseur qui lui présente l'interface voulue.*

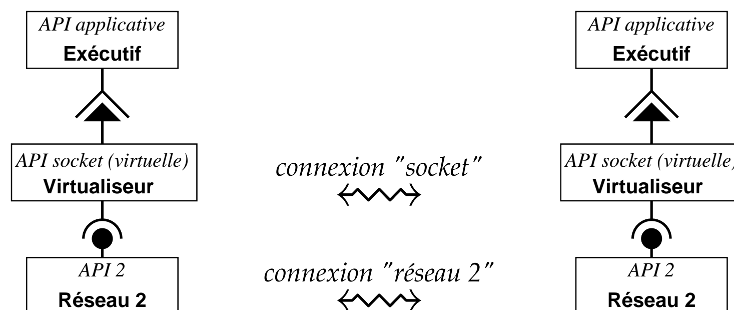

FIG. 7.2 – *Connexions virtuelles entre les couches logicielles.*

Pour pouvoir réutiliser les exécutifs existants sans les modifier, nous les considérons comme des *composants*, entités à part entière dont l'accès se fait uniquement au travers d'interfaces dont l'utilisation est régie par un contrat entre les deux parties : ils fournissent un certain service, et utilisent d'autres services. Les services fournis sont destinés à être utilisés par une application ou un environnement de plus haut niveau ; nous ne modifions pas cette partie. Les services utilisés (*uses* dans la langue des composants) sont pour la plupart des fonctions des bibliothèques systèmes. Pour qu'un exécutif utilise une ressource qu'il ne connaît pas, il suffit de lui fournir un accès à ce réseau sous la forme d'une API qu'il connaît.

La figure 7.1 représente un exemple de virtualisation des ressources réseaux pour un exécutif. Dans cet exemple, l'exécutif est conçu pour utiliser une interface de type *socket* (symbolisée par ▲). La partie gauche de la figure correspond à l'utilisation habituelle : l'exécutif utilise l'API *socket* du système pour accéder à TCP/IP par exemple. La virtualisation consiste alors en le remplacement de cette version *sockets* du système par une autre implémentation qui présente la même interface ▲, comme représenté à droite de la figure 7.1. Nous appelons cette autre implémentation un *virtualiseur*. Puisque l'API présentée à l'exécutif par le virtualiseur correspond à une émulation d'une API au-dessus d'une autre API, nous qualifions cette interface de *virtuelle*. À l'image de ce qui est fait dans le modèle OSI [37], d'un point de vue virtuel chaque couche dialogue avec son homologue à l'autre extrémité de la connexion. Le virtualiseur est donc totalement libre de réaliser l'acheminement comme bon lui semble, du moment que le comportement de l'interface est correct. Comme l'illustre la figure 7.2, les virtualiseurs aux deux extrémités de la connexion dialoguent par l'intermédiaire du protocole *2* pour acheminer les données selon une sémantique *socket*. L'exécutif ne "voit" pas le protocole *2* — donc ne voit pas la connexion de type *réseau 2* —, il ne voit que l'interface *socket* fournie par le virtualiseur. Que l'implémentation *socket* soit réelle ou virtuelle, l'exécutif utilise toujours l'interface *socket* et manipule des connexions *socket* sans avoir besoin de manipuler les notions des couches inférieures conformément au modèle en couches indépendantes.

L'utilisation de la virtualisation assure la portabilité d'un exécutif sur une nouvelle ressource ré-

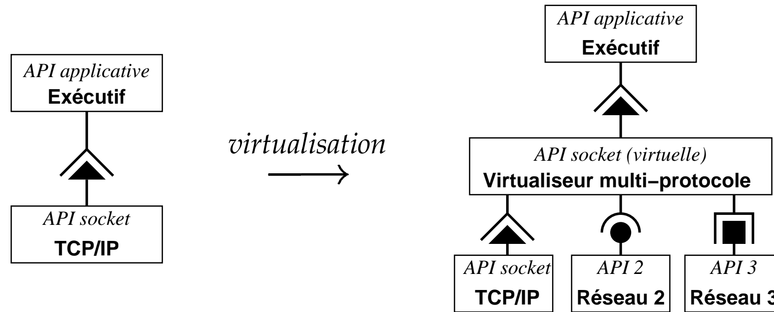

FIG. 7.3 – *La virtualisation permet une portabilité transparente.*

seau qu'il ne connaît pas, pour laquelle il n'était pas conçu au départ. L'intérêt principal réside dans le fait que cette portabilité est apportée sans avoir besoin de modifier les mécanismes internes des exécutifs. Puisque le virtualiseur a toute latitude en ce qui concerne l'implémentation réelle, nous pouvons imaginer des schémas plus élaborés que celui de la figure 7.1. Par exemple, il est possible de rendre le virtualiseur capable d'utiliser plusieurs types de réseaux, comme illustré à la figure 7.3. Sur cet exemple, l'exécutif ne voit que l'interface ▲ du virtualiseur, et devient donc d'emblée portable sur les réseau 2 et réseau 3 accessibles par les interfaces ● et ■.

Les mécanismes de virtualisation permettent donc de voir un exécutif comme un composant logiciel : il fournit des services, en utilise d'autres, et sa composition avec les composants qu'il utilise est gérée par une tierce partie. En lui fournissant les services dont il a besoin implémentés sous une forme inhabituelle, nous sommes alors capable de lui faire utiliser différents types de réseau. Un virtualiseur multi-protocole permet une utilisation transparente de plusieurs types de réseaux.

7.3 Virtualisation par personnalités

Cette section étudie l'intégration des mécanismes de virtualisation dans l'architecture globale du modèle de plate-forme décrit à la section 5.2. Le but est de pouvoir réutiliser des exécutifs existants de façon transparente, y compris lors de traduction trans-paradigme. Dans le chapitre 6 qui décrit le modèle d'abstraction, nous avons proposé un modèle d'abstraction *multi-paradigme* qui ne fait que les compromis nécessaires, c'est-à-dire uniquement lors de l'utilisation de passerelles trans-paradigme. Nous avons montré que ce modèle d'abstraction permet un bon compromis entre les fonctionnalités présentées aux exécutifs et la complexité des adaptateurs à développer. L'utilisation des exécutifs existants sur les ressources des grilles consiste donc en l'utilisation de l'abstraction adéquate de ce modèle par l'exécutif.

7.3.1 "Portabilité" d'une abstraction sous n exécutifs

La solution qui vient immédiatement à l'esprit est d'utiliser l'abstraction choisie comme une interface générique. Ceci se traduit en pratique par la modification de tous les exécutifs susceptibles de l'utiliser pour qu'ils utilisent plutôt l'interface abstraite choisie. Avec cette approche, le portage de n exécutifs sur l'abstraction requiert la modification interne des n exécutifs, ce qui est une tâche fastidieuse et sujette à erreur.

Nous remarquons qu'il s'agit d'un problème très similaire au problème de la portabilité introduite à la section 2.1.3.2 page 12. Dans le problème de portabilité, il était question de disposer d'un exécutif sur n réseaux différents, ce qui était atteint en approche basique par la réalisation de n variantes du même code. Le problème est semblable à une symétrie près — exécutifs et réseaux sont inversés. Nous choisissons donc d'appliquer le même type de solution — à savoir, l'utilisation de *pilotes*. Pour n exécutifs sur une abstraction, ceci revient au développement de n codes d'adaptation entre l'interface abstraite présentée par la plate-forme de communication et l'abstraction utilisée en interne dans

l'exécutif. Ces n codes d'adaptations sont plus légers et donc plus faciles à écrire qu'une modification en profondeur des exécutifs.

Dans un deuxième temps, la méthode d'abstraction des ressources pour la portabilité avait abouti au principe de *généricité*. La généricité évite le redéveloppement de plusieurs codes similaires dans le cas de m exécutifs sur n réseaux ; elle préconise de plutôt proposer une interface générique utilisable par plusieurs logiciels différents pour éviter le développement de $m \times n$ pilotes. Si on retourne ce principe en vertu de la symétrie décrite ci-dessus, il s'agit alors de ne pas lier un exécutif à une plate-forme de communications particulière. Pour n exécutifs et m plates-formes de communications, il faudrait développer $n \times m$ codes d'adaptation entre plate-forme de communications et exécutif. La généricité dans ce cas consiste donc en la possibilité d'utiliser la même version d'un exécutif sur plusieurs plates-formes de communications ou directement sur le système d'exploitation, sans avoir besoin des différents codes d'adaptation.

Le problème de la portabilité d'une abstraction sous n exécutifs revient donc à dégager une API générique qui ne soit pas spécifique à une plate-forme en particulier. Une telle API est un point de convergence de deux aspects de généricité :

- généricité des plates-formes de communication : l'API présentée par la plate-forme est susceptible d'être utilisée par plusieurs exécutifs ;
- généricité des exécutifs : l'API présentée par la plate-forme est susceptible d'être présentée par plusieurs plates-formes.

La section suivante étudie le choix d'une telle API.

7.3.2 Quelle API générique ?

Conformément au modèle multi-abstractions, une plate-forme de communication fournit une API générique orientée vers le calcul réparti et un autre orientée vers le parallélisme. Pourtant, même en séparant les interfaces selon leur paradigme, il semble improbable de trouver une unique API en réparti et une unique API en parallèle qui satisfasse les deux critères de généricité simultanément en s'appliquant à tous les exécutifs à l'intérieur d'un paradigme. En effet, la généricité d'un exécutif consiste en la réalisation d'une unique version d'un exécutif, sans avoir besoin d'innombrables pilotes pour les différents types de réseaux. Une seule API permet d'assurer la portabilité sur différentes plates-formes de communications.

En réparti, il est indubitable que l'API *socket* Berkeley [38], sûrement l'interface réseau la plus populaire, est utilisée par de nombreux exécutifs. Elle serait donc candidate pour être cette API unique pour la généricité des exécutifs en réparti. Cependant, au vu de l'étude de la section 2.3.3, d'autres interfaces existent et sont susceptibles d'être utilisées par les exécutifs, comme par exemple XTI, Posix AIO, ou les récentes propositions pour des entrées/sorties à haut niveau de concurrence.

En parallélisme, aucun standard ne se distingue vraiment au niveau abstrait : *Fast Messages*, *Active Messages*, Madeleine, Gamma, ce qui en amène certains à utiliser MPI faute de mieux dans ce rôle pourtant inapproprié.

Cependant, ces nombreuses API recouvrent des notions relativement proches. Même s'il est possible de regrouper les propriétés de la plupart des plates-formes de communications sous la forme d'une interface abstraite dans chaque paradigme, il ne semble pas exister à l'heure actuelle d'API unique qui fédérerait tous les exécutifs d'un paradigme.

7.3.3 Notion de personnalité

Puisqu'il ne se dégage pas de standard d'API universelle ni en réparti, ni en parallèle, alors nous proposons de présenter aux exécutifs l'interface qu'ils attendent lorsqu'ils sont déployés sur leur réseau de prédilection. À l'intérieur d'un paradigme, ces différentes interfaces sont suffisamment proches pour que l'adaptation au-dessus de l'interface abstraite soit mince. Nous appelons ces minces codes d'adaptation d'interface des *personnalités* : comme l'illustre la figure 7.4, ils fournissent simplement plusieurs API (des personnalités) différentes à une abstraction. Il est donc naturel qu'il y ait plusieurs personnalités sur chacune des abstractions pour s'adapter au différents exécutifs supportés.

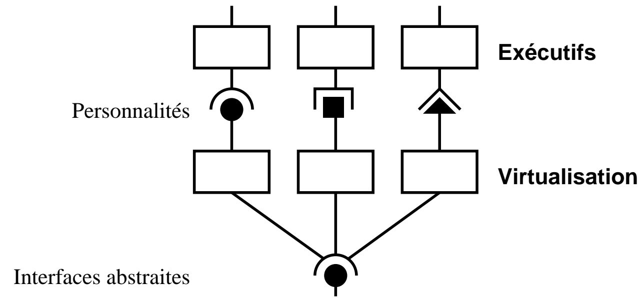

FIG. 7.4 – *Virtualisation par personnalités : présenter diverses interfaces au-dessus d'une abstraction en fonction des exécutifs — les différents symboles représentent des interfaces différentes, mais relativement proches.*

Cependant, leur nombre devrait être limité. En effet, une personnalité est susceptible d'être utilisée par plusieurs exécutifs. Par exemple, des implémentations CORBA et SOAP partagent la personnalité *sockets* au-dessus de l'abstraction répartie.

Le principal avantage de la virtualisation par personnalité est de ne demander *aucune* modification aux exécutifs puisqu'on leur fournit une autre implémentation d'une interface qu'ils utilisent déjà par ailleurs. Par conséquent il est aisé de suivre les versions successives des différents exécutifs. La prise en compte de nouveaux exécutifs est immédiate s'ils utilisent une personnalité déjà implémentée pour un autre exécutif. Dans le cas contraire, l'écriture d'une nouvelle personnalité — adaptateur fin qui traduit uniquement l'API, pas le paradigme — est une tâche aisée et rapide. Ceci ne crée donc pas de "branches de développement" dans les exécutifs, branches toujours difficiles à maintenir. Seule est utilisée la "branche principale".

7.4 Transparence

Comme nous venons de le voir, il est primordial d'intégrer des exécutifs existants. Nous avons proposé de virtualiser l'accès aux ressources de communication pour intégrer des exécutifs sans les modifier. Cependant, pour fonctionner cette démarche nécessite une *transparence* totale. Le virtualiseur doit se comporter exactement comme ce qu'il simule, aussi bien du point de vue de l'interface que du protocole ; comme l'illustre la figure 7.5, il doit offrir :

- transparence d'utilisation (transparence "vers le haut"), c'est-à-dire que l'ensemble adaptateur + personnalité se comporte du point de vue de l'utilisateur de l'interface exactement comme l'implémentation réelle qu'il émule ;
- transparence de protocole (transparence "vers le bas") : dans le cas du réparti où l'interopérabilité est nécessaire, il y a aussi besoin que l'ensemble adaptateur + personnalité se comporte comme ce qu'il émule du point de vue du système.

Nous décrivons dans la suite les aspects liés à ces deux formes de transparence.

7.4.1 Transparence d'utilisation

La transparence *vers le haut* consiste en un comportement identique à l'implémentation simulée *vue d'en haut*, c'est-à-dire du point de vue de l'utilisateur de l'API. Ceci implique d'une part d'émuler parfaitement le fonctionnement lorsque les ressources réelles sont différentes des ressources émulées, et d'autre part d'automatiser entièrement les mécanismes supplémentaires mis en jeu.

Transparence de comportement. L'utilisateur de l'API — un exécutif — ne "sait" pas qu'il utilise une implémentation qui virtualise la ressource. Il l'utilise exactement comme s'il s'agissait de la ressource réelle. En reprenant l'exemple de la figure 7.1, l'exécutif utilise l'API *socket* (représentée par ▲), censée être utilisée sur TCP/IP ; quand il utilise une personnalité qui présente l'interface *socket* mais sur un

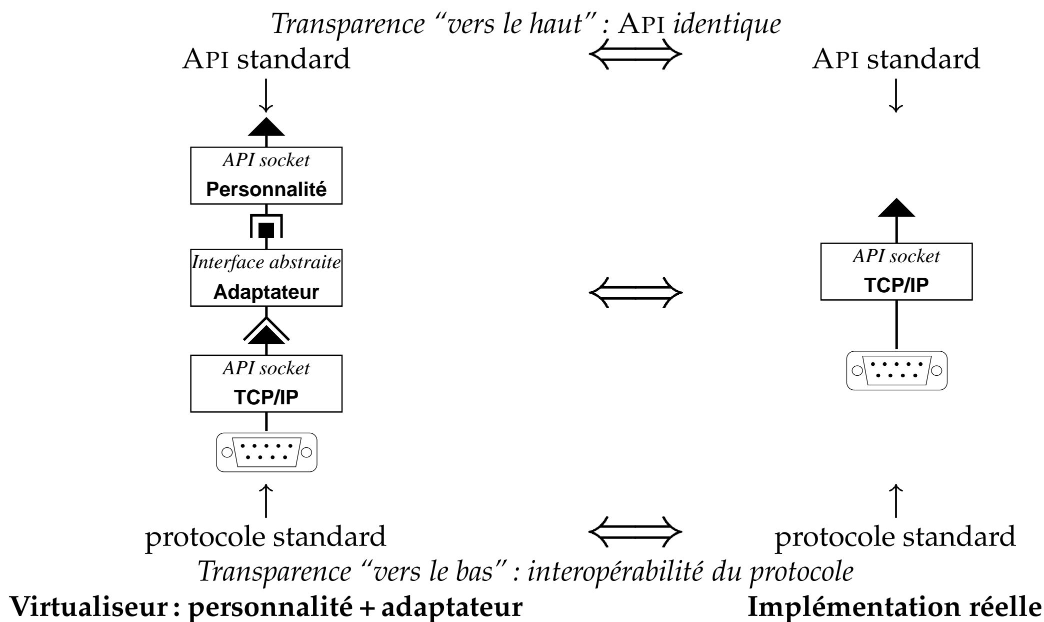

FIG. 7.5 – *Transparence : l'ensemble personnalité + abstraction se comporte comme l'implémentation réelle.*

autre réseau ("réseau 2" dans l'exemple), l'interface *socket* doit réagir exactement comme si TCP/IP était réellement utilisé en dessous, jusque dans les moindres détails de la norme, y compris dans les cas où l'utilisateur appelle des opérations qui n'ont aucun sens lorsque TCP/IP n'est pas réellement utilisé — que signifie la taille de fenêtre TCP (SO_SNDBUF) dans le cas où l'on re-route par Myrinet ? De même, des personnalités *Fast Messages* ou Madeleine doivent réagir exactement comme les implémentations réelles de *Fast Messages* et Madeleine. Ces points techniques spécifiques doivent être pris en compte lors de l'implémentation.

Sélection automatique. La transparence de comportement est confrontée au problème de l'émulation de fonctionnalités exprimables par l'API mais qui n'ont éventuellement aucun sens pour le réseau sous-jacent. À l'inverse, l'implémentation virtuelle possède des fonctionnalités supplémentaires par rapport à l'implémentation réelle. Ces fonctionnalités ne sont alors pas exprimables par l'interface utilisée, et l'exécutif qui l'utilise n'est pas censé connaître ces fonctionnalités. Le cas le plus flagrant est celui des mécanismes de sélection des méthodes de communication ou du réseau. Quand l'interface de la personnalité ne permet pas d'exprimer un choix de réseau à utiliser, il se pose la question de savoir qui prend la décision d'utiliser un réseau plutôt qu'un autre. Il est impératif que du point de vue de l'exécutif, cette décision soit automatique, c'est-à-dire que ce n'est pas à l'exécutif de prendre une décision à propos d'une fonctionnalité dont il ignore l'existence. La sélection du réseau ou du protocole par le *sélecteur* (voir la section 6.6) doit donc être entièrement automatique. Elle doit être automatique aussi bien pour les connexions sortantes — choisir l'interface d'interopérabilité en route par défaut — que pour les connexions entrantes — choisir sur quel protocole *écouter* quand on crée un serveur.

Pourtant, une transparence à tout prix qui cacherait systématiquement tous les choix pourrait être une limitation dans certains cas. Même si le cas général doit fonctionner en sélection automatique, il est intéressant de laisser la possibilité aux exécutifs d'intervenir sur les préférences du sélecteur. Dans les cas où ceux-ci "savent" qu'ils utilisent une plate-forme de communication particulière qui présente des fonctionnalités supplémentaires, ils peuvent utiliser l'interface de configuration du sélecteur.

7.4.2 Transparence de protocole

Dans le cas du paradigme réparti principalement, la transparence doit aussi être vérifiée au niveau du protocole, c'est-à-dire qu'un exécutif qui fonctionne sur une plate-forme de communication particulière doit rester interopérable avec les autres implémentations des exécutifs qui ne connaissent pas la plate-forme. Bien souvent l'interopérabilité est basée sur un protocole *sur le fil*. Par exemple CORBA, SOAP, `http` ou encore RMI Java construisent leur interopérabilité au-dessus de TCP/IP. Or après virtualisation de la ressource, il est possible que l'exécutif croit utiliser TCP/IP alors qu'en réalité il utilise un réseau totalement différent, et bien sûr incompatible ! Comment conserver l'interopérabilité avec des clients qui utilisent une implémentation CORBA standard dans ce cas ? Le problème peut être scindé en deux :

– la sélection de protocole interopérable : considérons le cas de la figure 7.3, où la virtualisation cache plusieurs protocoles. La plate-forme doit être en mesure de sélectionner le protocole ▲ quand un exécutif est basé sur la personnalité ▲.
– la transparence sur le fil : considérons le cas décrit à la figure 7.5, où le protocole adéquat a été sélectionné ; l'ensemble du virtualiseur ne doit pas interférer avec le protocole sur le fil, d'un bout à l'autre. Le pair distant n'est pas en mesure de supporter le moindre changement dans le protocole sur le fil. Dans ce cas, toute la plate-forme (personnalité, adaptateur, arbitrage) doit se comporter exactement comme une utilisation directe du protocole, y compris du point de vue du système.

La transparence sur le fil est obtenue en prenant en compte cette considération lors de l'implémentation de chacun des composants potentiellement utilisé lors de communications interopérables.

La sélection de protocole interopérable doit être gérée par certaines règles du sélecteur. Pour ne pas perdre en généralité, nous ne voulons pas lier les mécanismes de sélection automatique avec un protocole particulier. Il est donc préférable que le protocole par défaut soit choisi par la personnalité : quand la sélection automatique ne peut pas décider, la personnalité est la seule à savoir quel protocole elle adresse habituellement.

Dans l'exemple de la figure 7.3, l'interopérabilité de l'exécutif est assurée par le protocole TCP/IP, utilisé *via* l'interface *socket* symbolisée par ▲. L'interface *socket* servant habituellement pour TCP/IP, si la sélection automatique échoue (le nœud distant est inconnu), la personnalité ▲ demande explicitement TCP/IP au sélecteur. Ces mécanismes ne suffisent pas pour écouter les connexions en tant que serveur. La plate-forme n'est pas totalement libre dans ses choix d'adaptateurs et réseaux à activer : elle doit également accepter des connexions de la part de pairs distants qui n'utilisent pas le sélecteur. Par exemple, si la plate-forme juge que le réseau 2 est préférable à TCP/IP, elle ne peut pas se contenter d'activer uniquement le réseau 2, car l'exécutif n'est alors plus interopérable en tant que serveur avec ses versions standard qui fixent le protocole d'échange comme étant TCP/IP. Pour les connexions entrantes en mode d'interopérabilité, il faut impérativement être prêt à recevoir des connexions par le protocole qui lui sert de support. Par exemple, pour un exécutif basé sur l'API *socket* et qui a son interopérabilité basée sur TCP/IP, même si les connexions entrantes sont acheminées par le canal de contrôle et arrivent directement au sélecteur quand le client utilise notre plate-forme, il faut créer une *socket* TCP/IP en écoute pour recevoir les connexions venant de nœud qui utilisent une implémentation standard qui ne connaît que la connexion TCP/IP directe et ne connaît pas le sélecteur.

Il existe des cas plus problématiques. Considérons par exemple le cas d'une machine parallèle dont seul un nœud frontal est accessible par IP, et d'un exécutif dont l'interopérabilité est basée sur TCP/IP et lancé sur un des nœuds ne disposant pas de TCP/IP. Même s'il est possible de présenter à cet exécutif une API *socket* virtuelle (transparence de comportement), il semble plus problématique d'assurer une transparence de protocole, dans la mesure où le protocole voulu n'est pas disponible sur le nœud. Dans ce cas, il est certes possible de maintenir une interopérabilité grâce à un mécanisme de passerelle qui déporte l'utilisation de TCP/IP vers le nœud qui en dispose, et en ré-acheminant les messages par le réseau interne de la machine. Cependant, ces cas sont marginaux. Il semble qu'assurer l'interopérabilité lors de l'utilisation d'une personnalité sur un système qui ne dispose pas d'implémentation native du service habituellement fourni par la personnalité nécessite une prise en charge au cas par cas, dépendante des propriétés du protocole particulier, et donc au-delà de la portée de la

virtualisation en elle-même.

7.5 Discussion sur la virtualisation

La virtualisation par personnalités établit des liens entre quatre notions présentées jusqu'alors : portabilité, généricité, virtualisation, et abstraction. Nous pouvons voir les trois premières comme des raffinements successifs du mécanisme général qu'est l'abstraction :

– la portabilité par pilotes permet à un exécutif d'utiliser différents réseaux au travers d'une API spécifique ;
– la généricité étend ces mécanismes à plusieurs exécutifs en les implémentant sous une forme plus générale donc réutilisable d'un exécutif à l'autre ;
– la virtualisation par personnalité y ajoute la possibilité de réutiliser ces mécanismes même pour des exécutifs qui ne sont pas prévus pour utiliser une interface générique, et sans les modifier.

Une virtualisation au cas par cas, telle que présentée initialement à la figure 7.1 page 95 est une adaptation spécifique pour chaque exécutif. Il s'agit d'un problème symétrique à l'adaptation au cas par cas sur chaque type de réseau. Là où des *pilotes* fournissent une convergence de bas en haut des différents types de réseaux vers une interface donnée, nous avons proposé l'emploi de *personnalités* qui établissent une convergence de haut en bas, des différents exécutifs sur une interface donnée.

Ceci introduit également une interaction entre le modèle d'abstraction proposé au chapitre 6 et le mécanisme de virtualisation par personnalité. En effet, nous avons proposé un modèle d'abstraction basé sur des interfaces abstraites, chaque interface étant implémentée sur chaque type de matériel par des *adaptateurs*. Nous pouvons donc considérer qu'un ensemble adaptateur + personnalité est un virtualiseur particulier pour une API donnée sur un type de réseau donné ; les différents assemblages de personnalités et d'adaptateurs représentent ainsi les combinaisons de toutes les API supportées sur tous les réseaux supportés, y compris des combinaisons trans-paradigme. La virtualisation par personnalités permet donc une relative automatisation (ou du moins une systématisation de la méthode) de l'adaptation d'une plate-forme de communications aux exécutifs existants.

7.6 Conclusion

Étant donnée la profusion des exécutifs que l'on aimerait utiliser sur les grilles de calcul, il ne semble pas raisonnable de demander une modification en profondeur de chaque exécutif pour pouvoir le déployer sur une grille. Nous avons pris le parti d'utiliser directement les codes existants, de façon à permettre l'utilisation d'exécutifs arbitraires même s'ils ne sont pas conçus pour une plate-forme de communication en particulier. De cette façon, ni les exécutifs ni les applications ne deviennent dépendants d'une plate-forme de communication. Nous avons proposé d'utiliser des mécanismes de virtualisation qui permettent de faire utiliser à des codes existants d'autres ressources que celles pour lesquelles ils sont conçus. Comme nous l'avons vu, la principale contrainte de cette approche non-intrusive concerne la transparence : transparence à l'utilisation *via* l'API et interopérabilité *via* le protocole sur le fil.

La virtualisation par personnalités permet d'utiliser toute une variété d'exécutifs différents sur les modèles d'abstraction décrits précédemment, ce qui en fait une approche économe en développements logiciels, évolutive, adaptable à de nouvelles situations. La combinaison des possibilités apportées par le modèle d'abstraction proposé et de la virtualisation autorise l'utilisation *a priori* de n'importe quel exécutif sur n'importe quel réseau.

Chapitre 8

Accès arbitré aux ressources

Sommaire

8.1 Méthodologie d'accès arbitré 104
8.1.1 Contrôler les interactions par virtualisation 104
8.1.2 Intégration aux infrastructures réseau 105
8.1.3 Positionnement de l'arbitrage dans la pile logicielle 106
8.2 Mise en commun de la scrutation 107
8.2.1 Paramètres d'étude de la concurrence 107
8.2.2 Arbitrage entre accès concurrents à la même méthode 108
8.2.3 Réception par messages actifs 109
8.3 Mécanismes de multiplexage 110
8.4 Compétition et équité 111
8.4.1 Arbitrage des accès courts en mode rappel 111
8.4.2 Arbitrage global 112
8.4.3 Priorités 114
8.5 Harmonisation des actions globales 115
8.6 Discussion et conclusion 116

Après avoir étudié un modèle d'abstraction multi-paradigme au chapitre 6 et un mécanisme de virtualisation basé sur des personnalités au chapitre 7, nous présentons dans ce chapitre le niveau le plus bas de notre modèle de plate-forme de communication, à savoir l'accès arbitré aux ressources. La couche d'accès arbitré offre des mécanismes d'accès aux ressources qui sont efficaces et arbitrés, c'est-à-dire qu'ils prennent en compte les conflits d'accès et les problèmes de réentrance afin de pouvoir disposer de plusieurs exécutifs simultanément à l'intérieur d'un processus, et éventuellement sur le même réseau.

Nous avons étudié différentes méthodes pour disposer de plusieurs exécutifs en même temps à la section 4.2 : plusieurs paradigmes simulés sur un seul exécutif, juxtaposer plusieurs exécutifs, ou utiliser une plate-forme de construction d'exécutifs. Comme nous l'avons vu, aucune de ces trois solutions n'est réellement satisfaisante : aucune ne fonctionne immédiatement avec les propriétés souhaitées. La première et la dernière demandent de redévelopper tous les exécutifs ce qui représente un travail énorme à la pérennité douteuse. Cette approche est incompatible avec les conclusions de la section 7.1 sur la nécessité d'intégrer des exécutifs existants plutôt que de les ré-écrire spécifiquement pour une plate-forme de communications donnée. C'est pourquoi nous avons choisi la voie qui consiste à intégrer des exécutifs existants. La section 4.2.2 a étudié la juxtaposition pure et simple des exécutifs standard. Nous avons vu que cette juxtaposition est problématique dans de nombreux cas. Pour résoudre ce problème au cas par cas, il existe certaines intégrations spécifiques pour des combinaisons d'exécutifs données. Dans ces cas, il est également nécessaire de réaliser des développements *ad hoc* à la pérennité douteuse au fil de l'évolution des différents exécutifs, au risque de devoir se priver de

leurs mises à jour futures et de devenir rapidement obsolètes. Nous introduisons un mécanisme qui permet de juxtaposer des exécutifs standard sur une même ressource et qui ne soit spécifique ni à un exécutif particulier, ni à une combinaison particulière.

Arbitrer : quoi, pourquoi ? L'accès concurrent aux réseaux par plusieurs exécutifs n'est pas une chose qui va de soi, y compris pour les méthodes de communications directement fournies par le système d'exploitation comme par exemple TCP/IP utilisé par une interface *sockets*, XTI ou AIO. Ces interfaces sont conçues pour gérer les accès concurrents entre processus, mais rarement entre les différents *threads* à l'intérieur d'un unique processus. La plupart du temps, si les *threads* sont également gérés par le système, il n'y a pas de problème de réentrance. Cependant, les systèmes sont conçus dans l'optique où les différents accès concurrents aux réseaux à l'intérieur d'un processus sont réalisés par une seule application et sont donc coopératifs ; dans ce cas, les choix globaux ou les questions d'iniquité ne sont pas des problèmes si les différents accès sont réalisés en tenant compte les uns des autres. Ceci ne devient un problème que lorsque ces accès sont réalisés par différents exécutifs qui ne sont pas conscients de la présence des autres. De plus, de nombreuses méthodes de programmation des réseaux du parallélisme ne permettent tout simplement pas plusieurs utilisateurs en même temps. Ces conflits d'accès aux ressources concernent :

- les réseaux, aussi bien gérées par le noyau que par des bibliothèques en espace utilisateur ;
- le *multi-threading* ;
- les signaux Unix.

La suite de cette partie sur l'accès arbitré s'organise comme suit. Nous présentons d'abord la méthodologie utilisée pour l'accès aux ressources : un accès arbitré à des méthodes natives. Puis nous nous focalisons sur le cœur du principe d'arbitrage, à savoir la mise en commun de la scrutation. Nous étudions ensuite les mécanismes de multiplexage, puis nous nous intéressons à la gestion de la compétition et de l'équité, et à l'harmonisation des choix globaux. Enfin, nous discutons de la pertinence, du positionnement et du coût d'une telle approche.

8.1 Méthodologie d'accès arbitré

Dans cette section, nous présentons globalement la méthode d'accès arbitré aux ressources que nous proposons : virtualisation pour gérer les interactions en dehors des exécutifs de façon générique, utilisation de chaque réseau au mieux par sa méthode native, et arbitrage le plus bas possible dans la pile logicielle.

8.1.1 Contrôler les interactions par virtualisation

Même si les interactions entre exécutifs sont nombreuses, à plusieurs endroits, et pas toujours prévisibles, nous remarquons toutefois qu'il n'existe pas d'interaction directe entre les différents exécutifs. Lorsque l'on tente des les juxtaposer dans le même processus, les exécutifs ne se "voient" pas directement. Les seules interactions — donc les seuls risques de conflits potentiels — surviennent lorsque plusieurs exécutifs en même temps accèdent à une ressource en supposant chacun qu'ils sont les seuls à l'utiliser. Ces interactions sont donc toujours indirectes, c'est-à-dire que les conflits ont lieu uniquement au travers d'actions des différents exécutifs sur une même ressource. Il s'agit simplement d'une collision d'accès.

Partant de ce constat, il suffit alors de permettre à chaque exécutif d'utiliser les ressources comme s'il était seul pour résoudre les conflits posés par la juxtaposition. C'est pourquoi nous proposons de présenter à chacun des exécutifs une API d'utilisation des ressources qui se comporte exactement comme s'il était seul dans le processus à accéder aux ressources. Nous reconnaissons là le principe de la *virtualisation* déjà utilisé pour permettre à un exécutif d'utiliser d'autres ressources réseaux que celles qu'il connaît. Les interactions entre les accès concurrents — réentrance, compétition, choix global — peuvent alors être gérées à un niveau inférieur, en dehors des exécutifs. La virtualisation par

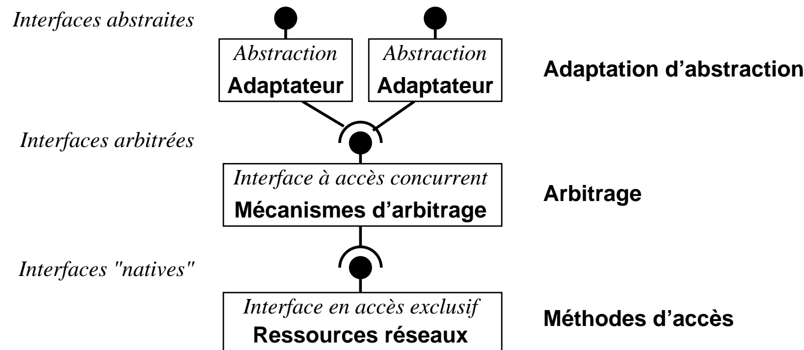

FIG. 8.1 – *Arbitrer l'accès aux ressources pour obtenir un accès concurrent : plusieurs adaptateurs peuvent utiliser la même ressource en même temps.*

personnalités et l'adaptation d'abstraction reporte le problème des accès concurrents sous les adaptateurs : plusieurs adaptateurs peuvent être amenés à accéder au réseau en même temps. Les mécanismes d'arbitrage sont regroupés dans une couche d'accès arbitré aux ressources qui s'interpose entre les ressources en accès exclusif et les adaptateurs qui veulent accéder à ces ressources, comme l'illustre la figure 8.1. Cette couche d'accès arbitré permet ainsi à plusieurs adaptateurs d'accéder en même temps à une ressource qui ne supporte pas l'accès concurrent. Les mécanismes d'arbitrage peuvent être vus comme un multiplexeur qui simule "plusieurs sur un".

8.1.2 Intégration aux infrastructures réseau

Les mécanismes d'arbitrage à mettre en œuvre sont largement dépendants des ressources utilisées et des propriétés des méthodes d'accès à ces ressources. Le but d'une couche d'accès arbitré au réseau est de fournir une méthode d'utilisation de chaque type de réseau de la meilleure façon possible, en supportant de multiples accès concurrents de façon à supporter plusieurs exécutifs, virtualiseurs ou abstractions selon le contexte. L'utilisation de chaque type de réseau *au mieux* passe par le choix d'une méthode d'accès qui est jugée *optimale* en fonction de certaines propriétés recherchées. À notre sens, l'utilisation "au mieux" des différentes ressources se fait au travers d'une méthode que l'on peut considérer comme *native*, c'est-à-dire la méthode qui représente le meilleur compromis entre la portabilité et l'adéquation à la ressource. Cette méthode peut être un accès direct, le passage par un pilote, ou l'utilisation d'une bibliothèque de communications qui apporterait déjà une certaine généricité. En particulier, les propriétés considérées pour ce choix sont :

performance — il est clair que dans le cadre de calculs numériques, et surtout en parallélisme, la performance du réseau effectivement exploitée est un critère déterminant.

expressivité — une méthode d'accès aux réseaux se doit de pouvoir exprimer pleinement les capacités du matériel, conformément à notre étude du modèle d'abstraction : les couches logicielles intermédiaires ne doivent pas perdre de propriétés du bas niveau dont le haut niveau pourrait avoir besoin. Le cas échéant, il vaut mieux définir plusieurs interfaces capables de rendre compte de toutes les propriétés plutôt que d'essayer de tout contraindre sur une seule. En particulier, il est préférable que la méthode d'accès soit conforme au paradigme pour lequel le matériel est conçu ; par exemple on utilisera Myrinet avec une méthode du parallélisme, et TCP/IP avec une méthode du réparti.

évolutivité — le choix éventuel d'une bibliothèque de communications ou l'utilisation directe d'un pilote système doit tenir compte des évolutions possibles, c'est-à-dire avoir la possibilité d'ajouter facilement le support pour un type de réseau.

complexité d'intégration — le choix d'une bibliothèque de communication pour accéder aux réseaux doit tenir compte de la facilité d'intégration de la bibliothèque dans un ensemble plus grand, et de son remplacement éventuel par une autre bibliothèque.

portabilité — la richesse de l'éventail de matériels supportés est cruciale pour que la diffusion du logiciel n'entraîne pas de coûteuses adaptations à d'autres types de matériel.

Méthode d'accès aux réseaux parallèles. Les technologies réseaux dédiées au parallélisme (SAN) sont nombreuses, sans réel standard qui s'impose. De plus, chaque type de réseau particulier s'utilise avec une méthode particulière. Pour obtenir de bonnes performances, il semble à première vue que le meilleur choix est d'utiliser chaque réseau directement avec la méthode fournie par le constructeur : GM pour Myrinet, SISCI pour SCI, IB Access pour InfiniBand, etc. Le risque majeur est alors la dispersion des efforts de développement pour simplement apporter la portabilité. Il semble donc pertinent de se tourner vers des bibliothèques de communications existantes du parallélisme, capables d'exploiter chaque réseau au mieux. La contrainte d'expressivité et celle liée de la performance incitent à ne pas chercher une bibliothèque de communications "à tout faire" mais plutôt une bibliothèque spécialisée pour le parallélisme et les réseaux parallèles. Par exemple Madeleine ou Panda semblent de bons candidats ; en particulier Madeleine obtient d'excellentes performances avec toujours plus de 90 % de la bande passante du matériel réellement exploitable. De cette façon, le matériel orienté vers le parallélisme peut être exploité à son maximum, sans perte de performance ou d'expressivité, tout en garantissant une bonne portabilité. Le caractère générique de ces bibliothèque dotées d'une interface "neutre" assure une relative indépendance vis-à-vis de la bibliothèque ; changer de bibliothèque de communication demande certes des adaptations, mais elles semblent mineures dans la mesure où les bibliothèques conçues pour le parallélisme ont des propriétés similaires : messages délimités, possibilité de zéro-copie, numérotation logique des nœuds, topologie statique suivant un modèle SPMD.

Méthode d'accès en réparti. Les technologies réseaux dédiées au réparti sont également très diversifiées, avec différentes API de programmation possibles. En revanche, même avec des interfaces de programmation et du matériel varié, il se dégage un standard d'interopérabilité : le protocole IP, fondation de la plupart des WAN. L'interopérabilité reste assurée même quand le protocole IP est utilisé au travers de différentes API — par exemple un client en XTI et un serveur en Posix AIO peuvent interopérer. La principale contrainte du réparti — à savoir l'interopérabilité — n'impose donc pas de supporter de nombreuses interfaces de programmation. Puisque les différentes interfaces fournies aux exécutifs sont incarnées par des personnalités, la couche d'accès arbitré a toute liberté quant au choix de l'interface de programmation à employer pour accéder aux réseaux du réparti. Dans le cas du réparti, il n'est donc pas nécessaire d'utiliser une bibliothèque de communications, du moins pas pour la portabilité.

Il existe des bibliothèques telles que ADAPTIVE, `liboop` ou `libevent` présentées au chapitre 2 qui facilitent la programmation asynchrone des réseaux IP. L'usage d'une telle bibliothèque ne semble toutefois pas pertinent dans le cas présent. De telles bibliothèques servent essentiellement à gérer une boucle de réception, ce qui est justement la partie que nous voulons contrôler finement. De plus, elles nous empêcheraient de contrôler précisément la gestion du *multi-threading* et les interactions entre les réseaux SAN et WAN.

Les méthodes d'accès aux réseaux optimales selon nos critères sont donc l'utilisation d'une bibliothèque de communication pour les réseaux parallèles, et l'utilisation directe des primitives fournies par le système pour les réseaux du réparti. Sans perdre en généralité, dans la suite nous supposons que ce sont les méthodes effectivement employées.

8.1.3 Positionnement de l'arbitrage dans la pile logicielle

Les différentes méthodes d'accès, bibliothèques génériques de communications ou appels directs au système, ont généralement besoin d'un arbitrage car elles ne fournissent pas toutes les propriétés nécessaires pour être utilisées par plusieurs codes en même temps de façon indépendante dans chaque processus. Il semble donc indispensable d'ajouter des mécanismes d'arbitrage ou de multiplexage *au-dessus* de ces méthodes d'accès.

D'un autre côté, notre modèle d'abstraction est bâti selon l'idée de proposer chaque paradigme à chaque niveau, c'est-à-dire que plusieurs interfaces abstraites sont susceptibles de cohabiter. Ceci se

traduit par l'utilisation simultanée d'un même réseau par plusieurs adaptateurs pour fournir les différents paradigmes au niveau abstrait au-dessus de toutes les ressources. Les différentes méthodes d'accès de bas niveau sont donc susceptibles d'être utilisées par plusieurs codes clients en même temps. Il faut donc que l'arbitrage et le multiplexage soient réalisés *aussi bas que possible* dans les couches du modèle. Pour que la gestion de la concurrence soit simple, performante et cohérente, il est donc souhaitable d'arbitrer immédiatement au-dessus de la méthode d'accès au réseau, de façon à construire un système qui supporte correctement la concurrence au-dessus.

Pour que l'arbitrage soit effectif, il est donc nécessaire que tous les accès aux ressources, à accès exclusif ou non, passent par la couche d'accès arbitré. Tout accès qui court-circuiterait les mécanismes d'accès arbitrés en s'adressant directement à la ressource mettrait en échec toute la politique d'arbitrage. Cependant, les accès aux ressources sont contrôlés finement : les adaptateurs et personnalités sont conçus spécifiquement pour une plate-forme données, donc utilisent les accès arbitrés dès leur conception ; les codes réutilisés tels-quels (exécutifs, applications) qui ne sont pas conçus pour un accès arbitré n'ont pas d'accès direct aux ressources car la virtualisation redirige tous leurs accès sur les adaptateurs, et donc au final sur l'accès arbitré en tous les cas. De cette façon, la couche d'accès arbitré est un *point d'accès unique* aux ressources ce qui permet de gérer la concurrence en prenant en compte tous les accès. La gestion de la concurrence s'organise selon trois axes :

réentrance — quand une ressource n'est simplement pas prévue pour être utilisée par plusieurs clients, la solution consiste à multiplexer les accès ("plusieurs sur un").

compétition — quand la concurrence est prévue mais que la juxtaposition de schémas d'utilisations différents mène à une iniquité d'utilisation, il faut alors contrôler l'entrelacement et gérer les priorités relatives.

choix global — certains choix globaux à un processus sont isolés par la virtualisation.

Les sections suivantes détaillent ces mécanismes un par un.

8.2 Mise en commun de la scrutation

Nous présentons dans cette section le principe de la mise commun des mécanismes de scrutation du réseau. Ce principe est le cœur du niveau d'accès arbitré ; il sera utilisé ensuite pour construire des mécanismes de multiplexage et gérer l'équité.

Gestion de la concurrence. Les principales ressources concernées par les problèmes de concurrence sont les réseaux — aussi bien ceux du parallélisme que ceux du réparti. Habituellement, les questions de réentrance sont résolues par une exclusion mutuelle. Cependant, les réseaux sont une ressource de nature différente des autres : pour assurer la réentrance, il ne suffit pas de garantir qu'un seul code à la fois accède au réseau ; il faut encore que chaque code utilise ses propres données ! Nous introduirons pour cela des mécanismes de multiplexage.

8.2.1 Paramètres d'étude de la concurrence

La gestion de la concurrence à ce niveau dépend fortement du degré de concurrence supporté par la méthode d'accès au réseau. Deux paramètres principaux jouent sur la stratégie d'arbitrage à employer : le *nombre de canaux* — ou encore *multiplexage natif* — fourni par la méthode d'accès et la *réentrance native* de la méthode d'accès. Lorsque le nombre de canaux fourni par la méthode d'accès est insuffisant, il doit être étendu par les mécanismes de multiplexage pour atteindre un nombre suffisant capable de supporter plusieurs exécutifs. Un manque de réentrance native est quant à lui comblé par des mécanismes d'exclusion mutuelle. Les deux paramètres peuvent varier indépendamment l'un de l'autre ; certaines combinaisons n'ont aucune correspondance avec la pratique et ne sont pas traitées ici. Nous distinguons quatre combinaisons majeures de ces deux paramètres :

1. accès exclusif — un seul canal de communication à accès exclusif, pas de réentrance ; exemple : SCI par Madeleine.

2. n canaux simultanés — plusieurs canaux utilisables simultanément en nombre n fixé ; exemple : Myrinet par Madeleine/BIP ($n = 2$ avec Madeleine 2, $n = 6$ avec Madeleine 3).
3. accès multiples non-réentrants — nombre arbitraire de canaux, mais réentrance non-garantie ; exemple : *sockets* BSD avec bibliothèque de *multi-threading* de niveau utilisateur (Marcel monoprocesseur).
4. réentrance assurée — nombre arbitraire de canaux utilisables simultanément sans problème de réentrance ; exemple : *sockets* BSD avec bibliothèque de *multi-threading* système (`pthread` ou PMarcel).

Les cas 1 et 3 ont besoin de mécanismes d'exclusion mutuelle. Les cas 1 et 2 ont besoin de multiplexage. Typiquement, les réseaux du parallélisme seront dans le cas 1 ou 2, les réseaux du réparti dans le cas 3 ou 4. Ces quatre cas de figure nous serviront d'exemple dans la suite.

Les mécanismes d'arbitrage destinés à assurer l'équité, la réentrance, et le multiplexage sont intimement liés. L'arbitrage est assuré par une infrastructure générale qui permet de gérer ces problèmes ensemble. Ces mécanismes sont basés sur trois entités :

scrutateur — entité qui assure la scrutation, c'est-à-dire l'observation du réseau pour connaître son état (émission possible, messages en attente, demande de connexion arrivée, etc.) ;

multiplexeur — entité qui offre plusieurs connexions logiques sur une seule connexion physique. Le multiplexeur utilise les services du scrutateur ;

arbitre — un scrutateur ou un ensemble scrutateur + multiplexeur selon le type de réseau, pour un réseau donné.

8.2.2 Arbitrage entre accès concurrents à la même méthode

Nous proposons tout d'abord un mécanisme d'arbitrage entre les différents accès concurrents à un même réseau. Il peut s'agir d'accès simultanés à TCP/IP ou à Myrinet par exemple. Le principe de l'arbitrage se traduit par une factorisation du code d'accès au réseau : tous les accès entrants ou sortants passent par la même méthode d'accès au réseau, donc vraisemblablement par le même code bas niveau. Il est alors à craindre que l'arbitre devienne un point de contention. Pour étudier l'éventualité d'une contention ou d'un goulot d'étranglement, nous distinguons trois catégories d'accès aux réseaux :

– les *accès à durée indéterminée*, tels que les réceptions depuis le réseau, les établissements de connexion entrante ou sortante, mais aussi dans certains cas les envois sur le réseau ; ce sont des accès dont la durée est entièrement dépendante des actions d'un autre nœud du système et peuvent *a priori* rester en attente indéfiniment.
– les *accès courts*, tels que la plupart des envois sur le réseau, quand on sait que les données peuvent partir ; leur durée n'est pas théoriquement bornée — il peut y avoir des artefacts de l'ordonnanceur et des collisions avec des communications en provenance ou à destination d'autres nœuds — mais le nœud distant n'est pas susceptible de provoquer une attente indéfinie.
– les *accès garantis*, qui sont des accès courts après obtention de l'exclusion mutuelle. Le nœud distant est prêt, mais la durée de l'opération peut cependant encore être influencée par des artefacts des ordonnanceurs de *threads* ou de processus, ou des perturbations sur le réseau.

Résolution des accès à durée indéterminée. Les accès à durée indéterminée posent un problème plus difficile que les accès courts. En effet, pour apporter la réentrance aux accès courts, il suffit de mettre en place un mécanisme d'exclusion mutuelle pour éviter les collisions d'accès, et de réaliser l'accès proprement dit en section critique. Pour les accès à durée indéterminée, il n'est pas envisageable de verrouiller la ressource pendant une durée indéterminée sans gêner le déroulement du programme. En effet, dans le cas des accès à durée indéterminée, plusieurs opérations peuvent être en attente simultanément — par exemple, plusieurs attentes de réception pour plusieurs exécutifs — et il n'est pas possible de savoir à l'avance laquelle terminera en premier ; si un exécutif en attente de réception verrouille le réseau, il bloque tout autre accès des autres exécutifs pendant une durée indéterminée, ce qui constitue une iniquité d'accès flagrante dommageable pour les performances

globales, voire peut mener à un interblocage. Plusieurs solutions sont envisageables pour arbitrer les accès à durée indéterminée : transformer les demandes d'accès simples en accès multiples ou confier la scrutation à un scrutateur permanent dans l'arbitre.

La première solution consiste en la scrutation de l'état d'avancement de toutes les opérations demandées par tous les clients de l'arbitre à chaque tentative d'accès ou scrutation de la part de n'importe quel client. Par exemple deux adaptateurs A et B souhaitent utiliser un même réseau qui ne supporte pas plusieurs clients en même temps ; nous interposons un arbitre entre A et B d'une part et le réseau d'autre part. Quand A et B attendent une réception en même temps et scrutent l'état du réseau, une scrutation déclenchée par A détecte l'arrivée d'un message, mais l'arbitre indique que le message est destiné à B ; on ne renvoie rien à A, et le message est délivré à B lorsqu'il demande une scrutation. Il est possible d'utiliser cette méthode avec une scrutation active aussi bien qu'avec une attente bloquante de la part des adaptateurs A et B. Cette méthode a l'avantage de ne scruter l'état du réseau que lorsqu'une opération à durée indéterminée est en cours, ce qui semble efficace. Cependant, elle présente un inconvénient majeur : si un message arrive alors qu'il n'est pas attendu, alors l'arbitre est bloqué si la méthode d'accès ne lui permet pas de procéder à la réception lui-même sans connaître le format du message entrant — c'est par exemple le cas de MPI (il faut connaître le type des données), Madeleine (il faut connaître le format d'empaquetage), ou UDP (il faut connaître la taille du datagramme). Un exécutif qui ne procéderait pas assez vite à la réception des données qui lui sont destinées bloquerait alors les accès au réseau des autres exécutif, en particulier en cas de multiplexage sur un seul canal. Cette stratégie d'arbitrage a un gros problème d'iniquité.

La deuxième solution pousse plus loin la séparation entre scrutation et l'accès proprement dit. Cette solution consiste en une scrutation permanente par l'arbitre lui-même. Nous entendons par "*permanente*" le fait que cette scrutation n'a pas lieu uniquement lorsqu'un adaptateur demande un accès, mais tourne en tâche de fond. La scrutation en tâche de fond peut ainsi être réalisée dans un *thread* dédié ou directement par l'ordonnanceur de *threads* quand celui-ci offre une telle fonctionnalité [65]. L'arbitre gère ainsi un scrutateur par réseau, avec un rythme de scrutation à déterminer que nous étudierons en section 8.4. De cette façon, il est possible de réaliser un schéma de réception bien adapté à la méthode d'accès au réseau et au système de *multi-threading* ; en effet, la scrutation — point critique pour la réactivité du réseau — est alors gérée en un unique point dans les couches basses de la plate-forme de communications, d'où une possible intégration fine avec les infrastructures de *multi-threading*. La scrutation réalisée indépendamment des demandes d'accès réels permet à ce mécanisme de ne pas subir l'influence du chargement ou du déchargement d'un adaptateur qui l'utilise. Cette méthode a l'avantage de détecter les messages aussi vite qu'ils arrivent. Cependant, ces mécanismes permanents risquent de gêner l'exécution des calculs en s'accaparant le processeur uniquement pour la scrutation ; l'intégration fine avec le *multi-threading* n'est donc pas simplement une possibilité mais une nécessité à prendre en compte pour garantir des bonnes performances du réseau tout en ne consommant pas trop de ressources.

8.2.3 Réception par messages actifs

La scrutation par l'arbitre, utilisée seule, ne résout pas totalement le problème du blocage du réseau quand l'exécutif récepteur n'est pas prêt à recevoir. En effet, le scrutateur peut ainsi détecter tout de suite la présence de données à recevoir, mais il ne peut pas procéder directement à la réception sans en connaître le format. Cependant, la scrutation dans l'arbitre permet à l'arbitre de prévenir l'adaptateur que des données lui étant destinées sont arrivées, afin qu'il les reçoive immédiatement pour éviter tout blocage du réseau pendant une durée indéterminée. Il s'agit donc d'un renversement de l'interaction habituelle arbitre-adaptateur : plutôt que d'attendre que l'adaptateur interroge le scrutateur pour savoir si des données lui étant destinées sont arrivées, le scrutateur appelle une fonction de l'adaptateur dès que des données sont arrivées. Cette solution est connue sous le nom de *messages actifs* (ou encore, *active messages*). En d'autres termes, l'adaptateur doit *toujours* être prêt à recevoir, de façon à pouvoir recevoir tout de suite quand la présence de données entrantes est détectée par la scrutation permanente. En cas de démultiplexage en particulier, le scrutateur ne sait pas à qui s'adressent les données avant d'avoir effectivement commencé la réception ; en appelant alors directement le code

capable de terminer cette réception, nous sommes assurés d'obtenir le temps de réception minimal, donc le verrouillage du réseau pendant un temps minimal.

La scrutation par l'arbitre associée aux messages actifs peut être vue comme une transformation des accès à durée indéterminée en attente passive puis accès courts quand le moment est opportun. L'obligation d'utilisation de messages actifs pour recevoir n'est dans ce contexte pas une très grande contrainte puisque les seuls utilisateurs de l'accès arbitré et donc du scrutateur sont les adaptateurs qui font partie intégrante de la plate-forme de communications. Il n'y a donc pas besoin de modifier les exécutifs pour prendre en compte cet aspect.

Généralisation. Nous remarquons qu'il n'est pas nécessaire de choisir entre la scrutation permanente par l'arbitre et la transformation des accès simples en accès multiples. Il est envisageable d'utiliser un modèle mixte qui combine les deux approches : le scrutateur gère la scrutation dans le cas général ; quand un adaptateur veut explicitement recevoir des données, il le signale au scrutateur pour forcer une scrutation immédiate dans le but d'améliorer la latence de réaction. Le forçage d'une scrutation immédiate correspond effectivement à un accès multiple provoqué par une demande d'accès simple. Au final, que ce soit directement ou par messages actifs gérés par le scrutateur, tous les accès sont ramenés à des accès courts déclenchés au moment opportun.

Ces mécanismes de scrutation par l'arbitre et messages actifs sont applicables aux quatre cas étudiés, quel que soit le degré de réentrance native et de multiplexage natif de la méthode d'accès. La scrutation commune n'est pas une fin en soi ; elle permet de gérer le multiplexage si besoin, et permet également un arbitrage fin pour assurer une équité dans l'accès aux ressources. En particulier, contrôler finement le rythme de scrutation de tous les réseaux — même de ceux réentrants nativement — permet de contrôler l'équité entre les différents réseaux et le niveau de concurrence souhaité. C'est pourquoi il nous semble pertinent d'étendre l'utilisation d'une scrutation commune à tous les accès. L'utilisation d'un mécanisme de scrutation permanent mis en commun pour le multiplexage et le contrôle de l'équité est décrit dans les deux sections suivantes.

8.3 Mécanismes de multiplexage

Principe du multiplexage. Pour simuler plusieurs connexions logiques sur un seul réseau, le mécanisme utilisé habituellement est le multiplexage. Le multiplexage consiste en la simulation de plusieurs ressources identiques sur une seule. L'usage du seul mot "*multiplexage*" est d'ailleurs abusif dans ce cas car en réalité nous pouvons distinguer deux problèmes distincts selon le sens considéré : le *multiplexage* lorsque plusieurs codes envoient sur le même réseau, et le *démultiplexage* lorsqu'un réseau reçoit des données à aiguiller vers plusieurs codes. Lors de l'envoi, chaque message est estampillé avec l'identifiant de la connexion logique — c'est le multiplexage —, puis tous les messages de toutes les connexions logiques empruntent le même canal physique. En réception, l'étiquette du canal logique accolée aux messages lors de l'envoi permet de retrouver à quelle connexion logique il correspond — c'est le démultiplexage — pour l'acheminer à son destinataire final. Ainsi par exemple les message d'un client CORBA ne parviennent pas à MPI en réception mais bien au serveur CORBA ! Le nombre de connexions logiques fournies par ce mécanisme dépend directement de la taille de l'espace de valeurs exprimables par l'étiquette. Typiquement, nous ajoutons une étiquette sur 16 bits ou 32 bits qui permettent respectivement plus de 65000 ou de 4 milliards de canaux logiques. Une taille de 32 bits devrait largement être suffisante pour l'utilisation que nous en faisons.

Agrégation d'en-têtes. Les mécanismes de multiplexage/démultiplexage sont surtout nécessaires sur les réseaux du parallélisme qui n'ont pas ou peu de multiplexage natif, mais qui ont souvent d'excellentes performances. Le multiplexage apporte une fonctionnalité, mais il ne faudrait pas que cet ajout soit au détriment des performances. Le multiplexage demande l'adjonction d'un estampillage de chaque message avec un numéro de canal logique. Deux possibilités s'offrent à nous : concaténer l'en-tête de multiplexage avec les données pour tout envoyer en un seul paquet, ou ajouter un paquet d'en-tête rien que pour le multiplexage. La première solution ne permet pas de recevoir les en-têtes

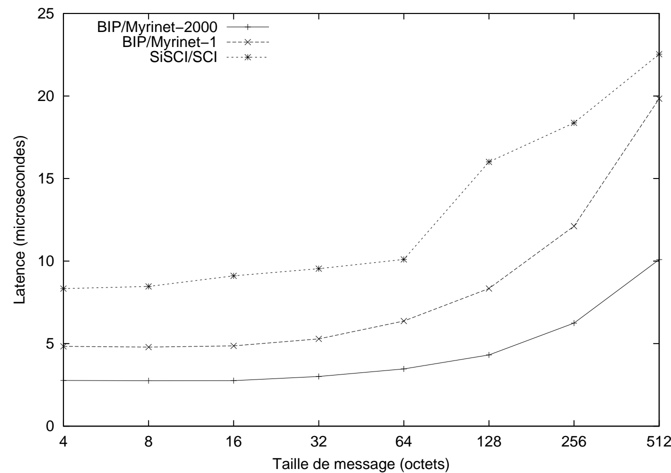

FIG. 8.2 – *Latence de transmission mesurée sur les* SAN *courants.*

et les données séparément ce qui risque de rendre impossible toute approche "zéro copie" et de nécessiter une copie en mémoire pour amener les données à leur emplacement final. En effet, avant de dépaqueter l'en-tête, on ne sait pas à qui les données sont destinées. Une copie en mémoire est catastrophique pour le débit sur un SAN. La deuxième solution, en envoyant systématiquement un paquet supplémentaire, est susceptible de multiplier la latence par deux, ce qui n'est pas satisfaisant.

Pourtant, en observant les schémas de communication des exécutifs, il apparaît que tous utilisent pour leurs échanges des messages en au moins deux parties : une partie d'en-tête pour identifier le message, et un corps de message contenant les données. L'en-tête est généralement petit (quelques dizaines d'octets au maximum) et de taille fixe pour un exécutif donné. Il est donc judicieux d'agréger l'en-tête du multiplexage avec l'en-tête que les exécutifs envoient de toute façon. Si l'étiquette de multiplexage est codée sur un entier 32 bits et qu'un exécutif utilise un en-tête de 24 octets, l'agrégation des en-têtes mène donc a des en-têtes de 28 octets sur le réseau et à une recopie en mémoire uniquement de ces $4 + 24$ octets et pas du corps du message. Ceci préserve alors aussi bien la latence que le débit global.

La figure 8.2 montre la latence mesurée sur quelques types courants de SAN à l'aide de Madeleine. Toutes les courbes présentent le même type de profil, avec une latence quasi-constante pour de petits messages. Il ressort très clairement qu'ajouter 4 octets à un en-tête de 24 octets qui aurait de toute façon été envoyé ne coûte presque rien, alors qu'envoyer un paquet de 4 octets puis un paquet de 24 octets prend environ deux fois plus de temps. Avec l'agrégation des en-têtes, le surcoût de latence dû au multiplexage se limite alors à quelques pourcents en utilisation réelle (quelques nanosecondes) ce qui est parfaitement acceptable et conforte notre choix de l'agrégation des en-têtes pour amortir le coût du multiplexage.

8.4 Compétition et équité

Nous avons proposé jusqu'à présent des mécanismes qui apportent la réentrance et le multiplexage. Cependant, quand plusieurs exécutifs utilisent plusieurs réseaux, une notion fondamentale doit être prise en compte dans la gestion de l'arbitrage : l'équité. Nous détaillons dans cette section les mécanismes mis en œuvre pour assurer l'équité lors des accès concurrents à un arbitre, et les interactions entre plusieurs arbitres sur plusieurs réseaux.

8.4.1 Arbitrage des accès courts en mode rappel

Tous les accès à durée indéterminée sont transformés par l'arbitre en *accès garantis*, effectués de façon asynchrone dans une fonction de rappel (*callback*, en anglais), appelée aussi *traitant*, quand la

scrutation donne le feu vert ; ces accès peuvent être une réception, un envoi ou une connexion. Nous décrivons ici la stratégie du scrutateur pour décider quand déclencher ces accès.

Nous appuyons notre étude sur les quatre cas définis à la page 107. Nous supposons que les méthodes qui ne fournissent pas nativement de multiplexage (cas 1 et 2) réalisent une scrutation par attente bloquante sur chaque connexion, et qu'en présence de multiplexage natif il est possible de réaliser des scrutations combinées pour plusieurs connexion en même temps à la façon des appels `select()` ou `poll()`, de façon bloquante ou non. Ces suppositions sont conformes à la réalité des méthodes d'accès considérées et ne limitent en rien la généralité de notre approche.

Sans réentrance native — Les combinaisons 1 et 3 n'offrent aucune réentrance. Le seul choix possible consiste en la mise en place de mécanismes d'exclusion mutuelle pour tous les accès. Aussi bien l'attente en scrutation que les accès courts ne peuvent être réalisé qu'un seul à la fois. Nous proposons donc d'utiliser une seule boucle de scrutation qui, en fonction du résultat des scrutations, appelle en séquence les différents accès courts. En l'absence d'information sur les communications futures, la meilleure stratégie à adopter est "premier arrivé, premier servi", c'est-à-dire que dès qu'une scrutation indique que l'accès est possible, l'accès est effectivement réalisé.

Avec n canaux natifs — Quand la méthode d'accès offre n canaux d'accès, alors nous mettons en place n boucles de scrutation pour utiliser toute la concurrence offerte par la méthode. Toutes les boucles peuvent alors réaliser des accès courts en même temps. Il faut alors réaliser un multiplexage "$m : n$" si m adaptateurs utilisent le réseau en même temps. Pour utiliser au mieux la concurrence offerte par la méthode d'accès, chaque adaptateur peut utiliser n'importe quel canal libre, et ainsi utiliser différents canaux d'un accès à l'autre. Pour chaque envoi, il faut alors allouer un canal sur lequel envoyer.

Avec réentrance native — Réaliser la scrutation dans une seule boucle évite un nombre de *threads* systèmes incontrôlable et des accès simultanés trop nombreux, synonymes de contentions. Les accès courts proprement dits sont ensuite répartis sur un *pool* de *threads* en nombre contrôlé.

Cette stratégie, avec les trois cas que nous venons d'exposer, présente deux caractéristiques importantes : les accès courts sont déclenchés uniquement au moment où le scrutateur a indiqué que le réseau est prêt, tout en maintenant un certain degré de concurrence quand la méthode d'accès le permet.

8.4.2 Arbitrage global

Nous désignons par *arbitrage global* les mécanismes qui gèrent la concurrence entre tous les accès, en particulier entre des accès à des réseaux différents. Nous avons jusqu'à présent présenté des mécanismes qui évitent des verrouillages à durée indéterminée pour chaque ressource : un ou plusieurs scrutateurs observent l'état courant de la ressource et déclenchent de façon asynchrone les opérations en attente quand elles peuvent être effectuées. Pour assurer une équité globale, il faut encore arbitrer entre les différents scrutateurs qui fonctionnent simultanément. Chaque scrutateur pouvant gêner les autres, l'approche se doit d'être *globale*, en tenant compte de toutes les ressources.

Entrelacement des scrutations. Concrètement, un scrutateur est incarné la plupart du temps par un *thread*. Un arbitrage global revient donc à agir sur l'exécution et l'entrelacement de ces *threads*. Le déroulement du *thread* dépend essentiellement des propriétés de la méthode de scrutation. Si la scrutation peut être effectuée de façon bloquante sans risque de blocage du processus, — cas de *sockets* avec des *threads* noyau ou de Madeleine avec des *threads* Marcel [65] —, alors l'arbitrage est directement pris en charge au niveau de l'ordonnanceur de processus ou de *threads*. Le scrutateur effectue en boucle (stratégie 1) :

1. scrutation de l'état du réseau ;
2. invocation des fonctions de rappel selon le résultat de la scrutation.

Si la scrutation n'est pas bloquante, alors une telle approche représente une attente active, ce qui n'est pas satisfaisant. Il faut alors réguler la vitesse de scrutation explicitement. Dans ce cas, le scrutateur

devient (stratégie 2) :

1. scrutation de l'état du réseau ;
2. invocation des fonctions de rappel selon le résultat de la scrutation ;
3. régulation de la concurrence (attente explicite ou retour à l'ordonnanceur).

La régulation est indispensable pour éviter une attente active qui ralentirait tous les autres *threads*, y compris les *threads* de calcul. La stratégie d'arbitrage est déterminée par l'attente. Un troisième cas se présente quand la scrutation n'est pas bloquante, mais que l'ordonnanceur de *threads* permet à l'utilisateur d'enregistrer ses fonctions de scrutation appelées périodiquement par l'ordonnanceur. C'est le cas par exemple de Marcel [65], avec la fonctionnalité dite "`marcel_poll`". Dans ce cas, la scrutation (étape 1 des deux stratégies précédentes) est appelée au rythme adéquat directement par l'ordonnanceur de *threads* ; elle mémorise les résultats de scrutation et débloque le *thread* de scrutation quand une opération est prête. Les opérations effectuées par le *thread* de scrutation sont alors (stratégie 3) :

1. attente du signal de la fonction de scrutation ;
2. invocation des fonctions de rappel selon le résultat de la scrutation.

Rafale et repos. En observant les schémas de communication courants, nous constatons que les communications ont souvent lieu en rafales — typiquement envoi ou réception d'en-têtes puis de corps de message éventuellement fractionné. Il est alors possible d'optimiser les scrutateurs basés sur la stratégie 2 (scrutation non-bloquante et attente de régulation) en détectant les rafales et en adoptant une stratégie plus agressive uniquement pendant la durée de la rafale. Pour offrir une bonne réactivité lors des communications en rafales sans pour autant adopter une stratégie agressive dans le cas général, nous distinguons deux états pour le scrutateur : rafale en cours (*burst*) ou au repos (*idle*). La détection du repos est effectuée en comptant le nombre de tours de boucle de scrutation successifs sans opération effective. Il est alors possible de proposer deux modes d'attente différents pendant les rafales et pendant le repos. Il est par exemple possible pour le scrutateur de garder la main pendant la rafale (pas de *yield*) et de la rendre seulement en fin de rafale.

Scrutation et ordonnancement. La politique d'ordonnancement est un autre critère primordial dans l'agressivité relative des scrutateurs. La plupart des ordonnanceurs de *threads* permettent au moins deux niveaux de priorités ; le plus haut niveau donne la main aux *threads* dès qu'ils sont prêts, même si d'autre étaient prêts avant.

En stratégie 1 (scrutation bloquante), le scrutateur peut être rendu prioritaire sans risque de gêner les autres *threads*. Rendre le *thread* prioritaire permet de borner la latence à au maximum le *quantum* de temps alloué à chaque *thread* par l'ordonnanceur dans le cas où aucun autre scrutateur n'est prêt en même temps. Quand le scrutateur est prêt, il est ainsi sûr d'avoir la main au plus vite. Le rendre non-prioritaire ne présente aucun intérêt, car la baisse de réactivité concédée par un niveau de priorité plus faible n'est contrebalancée par aucun avantage sensible. Le scrutateur passant l'essentiel de son temps en attente passive, en faire un *thread* prioritaire apporte une bonne réactivité sans gêner les autres *threads*.

En stratégie 2 (scrutation non-bloquante, attente de régulation), rendre le scrutateur prioritaire impose une attente avec un délai sous peine de famine des autres *threads*. Rendre ce délai égal à zéro revient à effectuer une attente active ; pour que la scrutation ne consomme pas trop de ressources, il est donc nécessaire que ce délai soit non-nul. L'usage d'un *thread* prioritaire assorti d'un délai assure que le scrutateur a la main à chaque *quantum* de temps de l'ordonnanceur, ce qui assure une consommation des ressources par le scrutateur constante, et une latence maximum de réception bornée même en présence de nombreux *threads* de calcul. Le niveau d'agressivité est réglé à l'aide du délai d'attente entre chaque scrutation. En pratique, la valeur du délai importe peu, car quand le *thread* a rendu la main, il la reprend au plus tôt un *quantum* de temps plus tard en présence d'autres *threads* actifs. Toutefois, abaisser le délai à zéro (*yield*) n'est pas intéressant car ceci oblige à rendre le scrutateur non-prioritaire, ce qui pénalise la latence en présence de nombreux *threads*, pour un gain minime.

En stratégie 3 (scrutation non-bloquante appelée par l'ordonnanceur), le *thread* qui réalise l'invocation des fonctions de rappel est bloqué et débloqué par la scrutation gérée en dehors du *thread*. Nous

avons donc tout intérêt à rendre ce *thread* prioritaire pour qu'il réalise effectivement les opérations dès que la scrutation a donné le feu vert. En effet, ce *thread* passe l'essentiel de son temps en attente passive à attendre le signal donné par la scrutation ; le rendre prioritaire ne pénalise donc pas les autres *threads* du processus.

En conclusion, à partir du moment où le rythme effectif de scrutation est contrôlé finement — par l'ordonnanceur en stratégies 1 et 3, par attente explicite en stratégie 2 —, nous ne voyons aucun inconvénient à rendre les scrutateurs prioritaires. Les fonctions de rappel invoquées par le scrutateur pouvant éventuellement dépendre d'actions réalisées par des *threads* moins prioritaires — cas fréquent si l'adaptateur utilise un mécanisme de boîte aux lettres —, il faut toutefois prendre garde aux problèmes d'inversion de priorité [168].

Bien qu'apparemment différentes, les stratégies 2 et 3 se ressemblent quand elles sont exécutées par un *thread* prioritaire : elles reviennent à effectuer une scrutation à chaque *quantum* de temps de l'ordonnanceur.

Régulation des scrutations forcées. Les scrutations forcées déclenchées par un adaptateur sont un risque d'iniquité. En effet, dans certains cas les scrutations forcées sont légitimes, mais il arrive qu'elle soient le signe d'un exécutif qui effectue une attente active. Il est alors nécessaire de réguler le rythme de scrutation forcée. Nous mettons en place un mécanisme de *quota* : au-delà d'un certain nombre de scrutations forcées dans un laps de temps donné, l'adaptateur qui a tenté une scrutation forcée abusive reçoit une pénalité sous forme d'attente avant que la scrutation ne soit effective.

8.4.3 Priorités

Nous venons de voir que l'arbitrage global n'est qu'un équilibre choisi dans l'agressivité relative de chaque scrutation. L'équilibre peut être éventuellement assuré dans le cas où l'ordonnanceur de *threads* supporte une telle fonctionnalité. Cependant, dans la plupart des cas l'amélioration de la réactivité d'un réseau entraîne une dégradation pour un autre réseau. Il est donc probable qu'en fonction des performances relatives des réseaux et des caractéristiques des applications, une unique stratégie d'arbitrage ne sera pas la meilleure dans tous les cas. La granularité relative des communications des différents paradigmes ainsi que l'impact de la latence sur l'application sont variables. Nous proposons deux niveaux où gérer les priorités : les priorités entre scrutateurs, et les priorités entre adaptateurs. Les priorités entre scrutateurs sont des priorités en fonction du réseau considéré ; les priorités entre adaptateurs sont des priorités en fonction des paradigmes des exécutifs.

Entre les différents scrutateurs, la gestion des priorités se fait en jouant dynamiquement sur les paramètres d'arbitrage présentés ci-dessus ; il est alors possible d'adapter la stratégie aux conditions. Il est donc essentiel de pouvoir adapter dynamiquement les paramètres d'attente de chacun des *threads* de scrutation. Cette façon de choisir les priorités, bien que simple et facilement compréhensible, n'est cependant pas adaptée à tous les cas. Par exemple pour une application parallèle à grains fins, il est préférable de donner la priorité aux communications issues de l'exécutif parallèle. Le choix dans ce cas ne se fait pas selon le réseau utilisé mais selon l'adaptateur qui déclenche l'opération sur le réseau. La priorité n'est donc pas attribuée à un scrutateur mais à certains accès aux scrutateurs. La façon de gérer ce type de priorité dépend du type de scrutateur :

- réseau sans réentrance native, avec multiplexage natif : une opération commencée ne peut pas être interrompue, et une seule opération peut être effectuée à la fois. La priorité est alors donnée en changeant l'ordre selon lequel les opérations sont effectuées, en réalisant toutes les opérations sur toutes les connexions prêtes à chaque tour ce qui élimine les risques de famine. Avec un multiplexage natif, à chaque tour de boucle, des opérations sur plusieurs connexions peuvent être réalisées ; nous choisissons l'ordre de traitement selon la priorité donnée, en traitant en premier les connexions les plus prioritaires.
- réseau sans réentrance native, sans multiplexage natif : il n'y a pas d'autre choix que de gérer les réception séquentiellement, dans l'ordre où les messages arrivent ; aucun ordonnancement ne

peut être fait en réception. En émission, une priorité peut être éventuellement être appliquée, en prenant garde toutefois d'éviter les famines.

– pour un réseaux à n scrutateurs (n canaux natifs, ou réentrance native régulée par n scrutateurs), ceux-ci sont chargés de l'accès au réseau. Un adaptateur prioritaire peut se réserver l'exclusivité d'un scrutateur sans risque de *deadlock* ni famine des autres, et obtenir ainsi un accès prioritaire. Cependant, ceci restreint le niveau de concurrence possible pour les autres adaptateurs.

8.5 Harmonisation des actions globales

Nous avons jusqu'à présent considéré l'accès arbitré uniquement du point de vue de la scrutation réseau, source de conflits et de compétition. Nous nous intéressons ici à la résolution des conflits liés aux *actions globales*, qui ont une influence sur tout le processus. Ces conflits surviennent lorsque plusieurs exécutifs effectuent des choix qui affectent tout le processus qui les contient. Habituellement, quand il est légitime de penser que c'est un choix qui n'aura aucun impact sur le code applicatif lui-même, cette approche fonctionne. À partir du moment où l'on tente de charger plusieurs exécutifs simultanément dans le même processus, les exécutifs peuvent ne pas être d'accord sur des réglages globaux qui affectent tous les processus, donc qui concernent tous les autres exécutifs chargés dans le même processus. Deux solutions sont mises en œuvre : la virtualisation, et le contrat de bonne conduite.

Résolution par virtualisation. La plupart des choix globaux sources de conflits peuvent être résolus par des mécanismes de virtualisation. La virtualisation permet à chaque exécutif d'avoir l'illusion qu'il effectue un choix global alors qu'il est local. L'environnement autour de l'exécutif réagit comme si le choix était global, mais en réalité il n'est pas propagé à tout le processus. De tels comportement globaux peuvent être isolés par virtualisation pour les ressources :

– pilotes réseaux : un exécutif peut vouloir utiliser spécifiquement BIP ou GM pour le réseau Myrinet. La virtualisation par personnalité permet de lui présenter l'API qu'il attend indépendamment de celle effective. Il est ainsi possible de faire cohabiter dans le même processus un exécutif basé sur GM, un autre basé sur BIP ; il est même alors possible de les faire utiliser autre chose que Myrinet !
– mémoire : la gestion de l'allocateur mémoire peut être modifiée. Par exemple BIP force le punaisage des pages, certaines bibliothèques de *multi-threading* demandent un verrou, etc. La plateforme gère l'allocateur mémoire en fonction des diverses contraintes et offre les services adaptés à chaque exécutif.
– signaux : les signaux Unix sont communs à tout un processus. Un traitant de signal mis en place par un exécutif peut être écrasé par celui mis en place par un autre exécutif. En isolant les traitants de signaux mis en place par les différents exécutifs, les actions globales n'entrent pas en collision.
– processus : un exécutif effectuant un `fork/exec` à l'insu des autres exécutifs du même processus risque de provoquer des erreurs. Nous détournons ces opérations vers des méthodes de chargement de code adaptées à la plate-forme. Toutefois, cette approche ne convient qu'aux cas où `fork` est immédiatement suivi d'un `exec` ; cette restriction n'est pas problématique dans la mesure où l'usage d'un `fork` seul est marginal. Les serveurs qui créent un processus par connexion (`fork` sans `exec`) doivent cependant être modifiés manuellement pour plutôt créer un *thread* par connexion.

Contrat de "bonne conduite". L'accès au *multi-threading* est de trop bas niveau pour pouvoir le virtualiser complètement. Beaucoup de fonctionnalités sont définies par du code `inline` incorporé dans le binaire à la compilation, et donc impossible à détourner par une virtualisation des symboles. Dans ce cas, il semble raisonnable d'imposer à tous les exécutifs d'utiliser la même bibliothèque de *multi-threading*. Le standard `pthread` tendant à se répandre, il est envisageable de demander à tous les

codes d'être basées sur l'interface `pthread`, même si au final l'implémentation peut être différente de celle du système comme celle fournie avec Marcel par exemple. Dans le cas d'utilisation d'une bibliothèque de *multi-threading* spécifique qui n'est pas compatible `pthread` au niveau binaire, nous utilisons une modification en amont, directement sur les sources. Cette modification pour convertir le code est toutefois réalisée automatiquement, sans intervention de l'utilisateur. C'est en quelque sorte une virtualisation qui agit au niveau du code source. C'est tout de même une virtualisation incomplète, car incapable d'agir sur des codes dont on n'aurait que les versions binaires.

8.6 Discussion et conclusion

Nous avons présenté des méthodes d'accès arbitrés aux réseaux. L'accès concurrent n'étant *a priori* pas garanti, le besoin se fait sentir de gérer les interactions entre les accès aux différentes ressources en même temps, et les interactions entre des accès concurrent à la même ressource. La gestion des ces interactions est grandement facilitée par la virtualisation qui assure que tous les accès aux ressources passent par la plate-forme de communication.

Nous pourrions nous poser la question de savoir si un arbitrage et une virtualisation à ce niveau ne fait pas double emploi avec le système d'exploitation, qui est habituellement l'entité qui gère ces problèmes. Nous pensons qu'il n'en est rien. Le but de notre approche est différent de celui d'un système d'exploitation. Plutôt que de chercher à isoler complètement les exécutifs et à gérer la sécurité entre eux, nous apportons simplement les mécanismes nécessaires pour les faire collaborer au lieu de se faire concurrence. Notre approche d'arbitrage et virtualisation de niveau utilisateur s'inscrit dans la même lignée que le *multi-threading* utilisateur par rapport au *multi-threading* noyau, et que les bibliothèques utilisateur d'accès au réseau (BIP par exemple) par rapport aux méthodes du système : avoir une plus grande liberté et s'affranchir des contraintes d'un OS (sécurité, gestion des permissions, isolement total, etc.) quand nous préférons un service moins strict et plus efficace.

Pour gérer les accès arbitrés, nous avons fait le choix de nous reposer sur l'existant — système d'exploitation pour IP, environnement générique pour les réseaux haute performance. C'est le choix de la *pérennité*, qui n'oblige pas à de coûteux développements logiciels pour la portabilité tout en permettant une exploitation efficace des ressources. Nous avons présenté des mécanismes de scrutation commune et permanente, associée à des messages actifs. Ce choix est raisonnable car il permet d'associer la gestion de la concurrence et du démultiplexage. Le principe des messages actifs a été introduit avec des environnements tels que *Active Messages* [178] ou *Fast Messages* [148], puis a été critiqué à cause des réceptions inopinées à n'importe quel moment qu'il provoque ; ceci pollue les caches d'instructions et de données du processeur et ralentit donc les calculs. Nous retenons le principe des messages actifs malgré tout car en environnement multi-exécutif, contrairement à ce qui se passe avec un seul exécutif, lorsqu'un message arrive on ne sait pas à qui il est destiné. Il est probable que dans ce cas, une approche sans messages actifs demande de toute façon un changement de contexte entre les primitives de démultiplexage et le *thread* applicatif à l'initiative de la réception. Le passage de flot d'exécution du démultiplexage à l'exécutif récepteur, qu'il soit réalisé par un changement de contexte de *threads* ou un message actif, perturbe dans tous les cas les caches du processeur. Le principal attrait des messages actifs est que la décision d'accéder au réseau est prise par la plate-forme, ce qui permet de gérer finement tous les accès, donc la concurrence.

Nous avons proposé pour gérer la concurrence plusieurs stratégies de scrutateurs en fonction des capacités de la méthode d'accès native. Le réglage de l'agressivité relative des différents scrutateurs agit comme un réglage de priorité. Le réglage de l'agressivité est contrôlé par des mécanismes que nous avons introduits : des attentes de régulation pour modérer l'agressivité, un mode rafale/repos pour favoriser les communications en rafale sans pénalité dans le cas général, et un forçage de scrutation avec *quota* pour une bonne réactivité sans consommation excessive de ressources même en présence d'attente active. La prise en compte des ordonnanceurs de *threads* avec une politique d'ordonnancement réglable permet de borner les attentes quand l'ordonnanceur propose cette fonctionnalité. Enfin, un mécanisme de priorités permet à une application de favoriser par exemple les communications issues de MPI ou de CORBA en présence d'un haut niveau de concurrence.

Troisième partie

Mise en œuvre et évaluation

Chapitre 9

La plate-forme PadicoTM

Sommaire

9.1 Présentation de PadicoTM **120**
9.1.1 Considérations générales de conception 120
9.1.2 Architecture générale de PadicoTM 120
9.2 Fondations de PadicoTM **121**
9.2.1 Micro-noyau : *Puk* 121
9.2.2 Gestionnaire de tâches : *TaskManager* 125
9.2.3 Accès arbitré au réseau : *NetAccess* 125
9.3 Le niveau abstrait dans PadicoTM **130**
9.3.1 L'abstraction pour le réparti : *VLink* 130
9.3.2 L'abstraction pour le parallélisme à mémoire distribuée : *Circuit* 133
9.3.3 Connaissance de la topologie et sélection automatique : *NetSelector* 135
9.4 Adaptateurs d'abstraction **140**
9.4.1 VLink/MadIO 140
9.4.2 Exemples d'adaptateurs généralisés 141
9.5 Personnalités et exécutifs sur PadicoTM **142**
9.5.1 Personnalités du réparti 142
9.5.2 Personnalités du parallélisme à mémoire distribuée 143
9.5.3 Exécutifs sur PadicoTM 143
9.6 Conclusion **144**

Dans ce chapitre nous décrivons la plate-forme de communication pour les grilles PadicoTM. Cette plate-forme met en œuvre le modèle que nous avons proposé dans la partie précédente. PadicoTM est l'exécutif de Padico. Padico [12] est une infrastructure logicielle pour la programmation des grilles de calcul. Padico est conçu pour le calcul parallèle, le calcul réparti, et les composants logiciels sur les grilles de calcul. Il cible les applications de couplage de codes basées sur le concept d'objets ou de composants CORBA parallèles. Padico est actuellement constituée de :

PaCO++ — l'infrastructure mettant en œuvre le concept d'objets parallèles ;

GRIDCCM — l'infrastructure mettant en œuvre le concept de composants CORBA parallèles ;

PadicoTM — la plate-forme de communications haute performance utilisée par PaCO++ et GRIDCCM.

Nous donnons tout d'abord un aperçu général de la plate-forme PadicoTM. Nous décrivons ensuite PadicoTM couche par couche, de bas en haut ; dans l'ordre : cœur, abstractions, personnalités, exécutifs.

9.1 Présentation de PadicoTM

PadicoTM est l'abréviation de *Padico Task Manager* — gestionnaire de tâches de Padico. C'est une plate-forme de communication pour grilles de calcul ; l'objectif principal est d'offrir des communications élargies à plusieurs paradigmes : plusieurs paradigmes dans les ressources utilisées, et plusieurs paradigmes dans les exécutifs proposés.

Cette section présente la philosophie de conception de PadicoTM et son architecture générale.

9.1.1 Considérations générales de conception

PadicoTM se doit d'être portable sur un grand nombre d'architectures, flexible et efficace. Il doit être capable d'intégrer des codes écrit avec différents langages : C, C++, FORTRAN, Java. Pour ces raisons, nous avons choisi d'écrire les fondations de PadicoTM en langage C. Étant donnée l'ampleur de la tâche à accomplir et la flexibilité voulue, nous avons fait le choix de la modularisation à l'aide de composants appelés "*modules*" dans PadicoTM. Les *modules* ont une interface définie, des implémentations interchangeables, et un assemblage dirigée par une tierce partie. Comme le propose la section 5.3, le chargement/déchargement des différents modules est entièrement dynamique pour intégrer les différents modèles d'exécution ; un *micro-noyau* est chargé de l'orchestration des modules. Tous les codes de PadicoTM sont encapsulés dans des modules, à l'exception du micro-noyau lui-même.

Plutôt que de redévelopper une infrastructure de portabilité complète sur les différents réseaux haute performance visés, nous choisissons d'utiliser un environnement générique existant comme proposé en section 8. Pour exploiter les réseaux "au mieux", nous tournons notre attention vers les environnements conçus initialement uniquement pour le parallélisme. Les environnements considérés sont notamment Madeleine, Nexus, Panda. Nous choisissons Madeleine pour ses bonnes performances, son large éventail de réseaux supportés, et la facilité de support technique que nous pouvons obtenir. De plus, Madeleine est particulièrement bien intégrée avec l'ordonnanceur de processus légers (*threads*) Marcel [138] pour tenir compte des interactions entre les *threads* et le réseau, ce qui rend possible une bonne réactivité à la réception. Nous choisissons donc de construire la partie basse de PadicoTM destinée au parallélisme sur Madeleine et Marcel.

Pour les réseaux du réparti, il est possible d'utiliser un environnement tel que ceux décrits à la section 2.3.3, par exemple ADAPTIVE, `libevent` ou `liboop`. Cependant, ni `libevent` ni `liboop` ne prennent en compte les aspects liés au *multi-threading* ce qui rend délicat leur utilisation en environnement *multi-thread*. Un environnement comme ADAPTIVE gère le *multi-threading* mais est essentiellement conçu pour apporter des abstractions C++ de haut niveau ; de plus, la cohabitation d'ADAPTIVE et Marcel est sans aucun doute problématique. Nous choisissons donc d'utiliser directement les *sockets* dans un module d'arbitrage qui suit la même logique que `liboop` mais en tenant compte du *multi-threading*. Ce n'est toutefois pas très pénalisant de repartir à zéro du niveau *socket* étant donné que `liboop` ne contient que 3 000 lignes de code.

9.1.2 Architecture générale de PadicoTM

La plate-forme PadicoTM est bâtie suivant le modèle décrit aux chapitres précédents. Elle est constituée des entités suivantes :

– *Puk*, un micro-noyau pour gérer les opérations de base sur les modules et les accès aux fonctionnalités fondamentales (allocation mémoire, bibliothèques dynamiques, etc.).

– un cœur ("PadicoTM *core*") constitué d'un jeu de modules de base ; ce sont les modules *TaskManager* et *NetAccess*, fondations pour la gestion des tâches et de l'accès réseau, et *NetSelector*, cœur du niveau abstrait, qui réalise la sélection et l'assemblage des adaptateurs.

– des modules optionnels, chargeables et déchargeables selon les besoins. Ce sont par exemple les adaptateurs, les personnalités et les exécutifs. Les applications sont également encapsulées dans des modules PadicoTM chargeables dynamiquement.

Les section suivantes de ce chapitre décrivent le micro-noyau et le cœur de PadicoTM, puis les adaptateurs, les personnalités, et enfin les exécutifs qui utilisent PadicoTM.

```
<defmod name="Console" driver="binary">
  <requires>PadicoTM</requires>
  <unit>Console.so</unit>
</defmod>
```

FIG. 9.1 – *Exemple de description de module : le module "Console".*

9.2 Fondations de PadicoTM

Dans cette section, nous décrivons les fondations de PadicoTM, qui sont constituées du micro-noyau *Puk*, du gestionnaire de tâches *TaskManager*, et de l'arbitre d'accès au réseau *NetAccess*.

9.2.1 Micro-noyau : *Puk*

Le micro-noyau de PadicoTM est appelé *Puk* — abréviation de *Padico µ-kernel*. Sa fonctionnalité principale est de gérer les opérations de base sur les modules. Le module est l'entité de base pour le chargement dynamique de code dans PadicoTM. Les modules sont chargés directement dans le processus Unix de *Puk* ; toute autre alternative engendrerait des surcoûts de communication élevés entre modules.

9.2.1.1 Modules *Puk*

Il existe différents types de modules, selon le type de code inclus (binaire, *bytecode* Java, script interprété, etc.), ou selon la sémantique des opérations sur les modules. Chaque type de module est géré par un *pilote* qui fournit les implémentations des opérations réalisables sur les modules de son type. La listes des pilotes de modules peut être modifiée dynamiquement en cours de session ; par exemple, le chargement d'un machine virtuelle Java provoque l'ajout dans la liste des pilotes de *Puk* d'un pilote pour gérer les modules qui contiennent du *bytecode* Java.

Un module est défini par un fichier de description en langage XML. La description de module contient les informations suivantes :

nom (attribut `name` de la balise `defmod`) — c'est le nom du module décrit.

pilote (attribut `driver` de la balise `defmod`) — c'est le nom du pilote *Puk* pour gérer le module.

dépendances (balise `requires`) — un module peut dépendre d'autres modules. Par exemple, un code applicatif utilisant CORBA et MPI dépend des modules `Corba` et `MPI`. Il s'agit d'une liste de modules dont *Puk* s'assurera qu'il sont chargés avant de charger le module voulu.

attributs (balise `attr`) — les attributs sont des paramètres de configuration du module. Un attribut est un couple composé d'une étiquette et d'une valeur, les deux sont des chaînes de caractères. Les attributs sont des valeurs associées à leur module respectif et peuvent être consultées à l'exécution.

unités (balise `unit`) — les unités sont le contenu "actif" des modules. Le contenu exact de chaque unité dépend du pilote ; par exemple, pour le pilote `binary` les unités sont des objets partagés chargeables dynamiquement, pour le pilote `java` les unités sont des fichiers `.class` ou `.jar`.

fichiers associés (balise `file`) — les fichiers associés sont le contenu "passif" des modules ; leur contenu est totalement libre pour le module lui-même. La seule opération réalisée par PadicoTM sur les fichiers associés est de les copier quand on copie un module.

La figure 9.1 montre un exemple simple de description XML, correspondant à un module appelé "`Console`" basé sur le pilote `binary`, ayant besoin du module `PadicoTM`, et étant implémenté sous la forme d'une seule unité binaire appelée `Console.so`. La figure 9.2 montre un exemple plus complexe correspondant à un module appelé "`CORBA-omniORB-4.0`", définissant un attribut, dépendant de trois autres modules et composé d'une unité binaire. Les deux fichiers associés, bien que fichiers binaires également ne sont pas des unités car ne sont pas destinés à être chargés explicitement ; ce sont plutôt des fichiers requis pour le fonctionnement de l'unité.

```
<defmod name="CORBA-omniORB-4.0" driver="binary">
  <attr label="NameService">
    corbaloc::paraski.irisa.fr:8088/NameService
  </attr>
  <requires>SysW</requires>
  <requires>VIO</requires>
  <requires>LibStdC++</requires>
  <unit>CORBA-omniORB-4.0.so</unit>
  <file>lib/libomnithread4.A.so</file>
  <file>lib/libomniORB4.A.so</file>
</defmod>
```

FIG. 9.2 – *Exemple de description de module : le module "CORBA-omniORB-4.0".*

```
  puk_mod_t puk_create (const char* module_desc);
padico_rc_t puk_load   (puk_mod_t mod);
padico_rc_t puk_start  (puk_mod_t mod, int argc, const char* argv[],
                        puk_job_t*out_job);
padico_rc_t puk_stop   (puk_job_t job);
padico_rc_t puk_unload (puk_mod_t mod);
padico_rc_t puk_destroy(puk_mod_t mod);
```

FIG. 9.3 – *L'interface de* Puk.

Opérations sur les modules. Les exécutifs, les bibliothèques et les applications sont des codes qui se comportent et se gèrent de façon différente. Certains ne servent que d'auxiliaire appelés par d'autres codes, alors que d'autres sont susceptibles d'incarner un fil principal (ou un fil secondaire en environnement d'exécution *multithreadé*. Pour tenir compte de ces différences, *Puk* comprend les opérations de base suivantes sur les modules :

`create` — il s'agit du chargement par *Puk* uniquement de la description XML du module, de façon à construire en mémoire la structure de description ;

`load` — *Puk* résout les éventuelles dépendances par un `create/load` sur chacun des modules requis s'ils ne sont pas déjà chargés, charge les unités en mémoire, et initialise le code du module proprement dit ;

`start` — par un appel à la fonction appropriée du pilote dépendant du type de module et intimement liées aux choix d'implémentation, *Puk* crée un fil d'exécution à partir d'un point d'entrée dans le module connu par convention (à la façon du `main()` en C/C++ et Java). Il est envisageable de créer plusieurs fils d'exécution pour un module s'il est réentrant.

`stop` — *Puk* demande au fil d'exécution de s'arrêter, par l'intermédiaire de la fonction appropriée du pilote. C'est une fonction similaire au `kill` Unix.

`unload` — *Puk* décharge le code du module de la mémoire ;

`destroy` — *Puk* détruit la structure de description du module en mémoire.

Cette interface est récapitulée par la figure 9.3 où les descripteurs de modules en mémoire ont le type `puk_mod_t` et les éventuelles erreurs sont codées dans des `padico_rc_t`. Chaque pilote fournit l'implémentation des primitives `load`, `start`, `stop` et `unload` sur une *unité* du type qu'il gère. Pour réaliser les opérations sur les modules, *Puk* appelle les opérations correspondantes des pilotes.

À son démarrage, avant de charger le moindre module, *Puk* connaît les pilotes suivants

`binary` — ce pilote est destiné au code binaire sous forme de bibliothèque dynamique, dans le format natif du système (ELF, Mach-O, etc.) ;

`pkg` — ce pilote gère des assemblages de modules (ou *packages*). Chacune de ses unités est elle-même un module.

`virtual` — ce pilote gère des modules sans unités, c'est-à-dire qui ne contiennent que des dépendances, des attributs, et des fichiers associés.

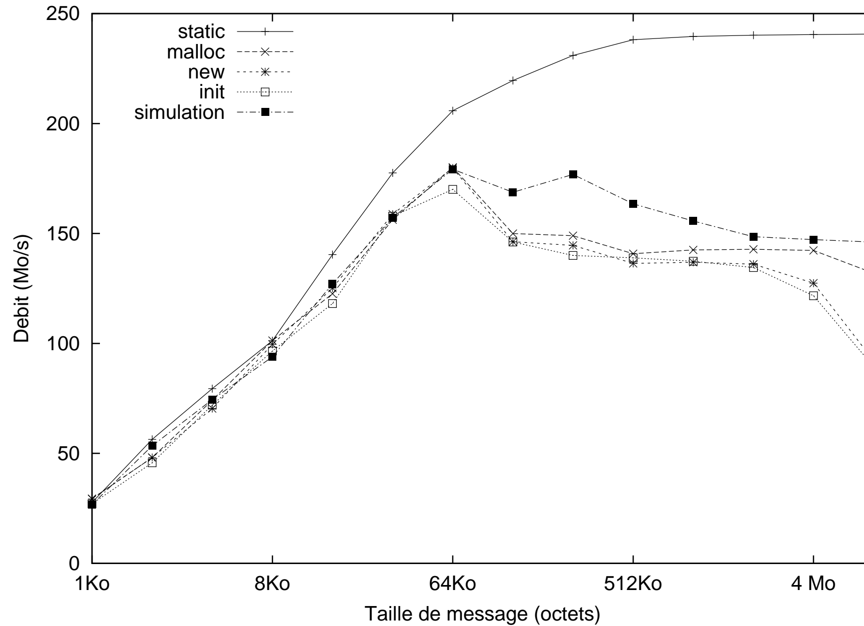

FIG. 9.4 – *Débit de transmission sur Myrinet-2000 en fonction de la stratégie d'allocation de la mémoire en réception*

9.2.1.2 Allocateur de mémoire semi-statique

Puk contient un allocateur de mémoire optimisé pour les réceptions depuis les réseaux rapides. En effet, ne faire aucune copie intermédiaire n'est pas suffisant pour exploiter entièrement le débit maximum permis par les réseaux haut débit. La méthode d'allocation de la mémoire à la réception est un autre facteur crucial. Un exécutif basé sur le concept de passage de messages reçoit, la plupart du temps, dans un tampon réutilisé d'une réception à l'autre. À l'inverse, un exécutif basé sur le concept d'invocation de procédure ou de méthode à distance reçoit les données dans une zone mémoire fraîchement allouée par le traitant de réception. Le fait de recevoir dans une zone déjà utilisée ou fraîchement allouée a une influence directe sur le débit obtenu.

Comparaison des allocateurs mémoire. Nous avons choisi pour mettre en évidence le problème le cas le plus marquant, à savoir Myrinet-2000, de débit nominal 250 Mo/s. Sur ce réseau dont le débit avoisine le débit de la mémoire elle-même, la stratégie d'allocation mémoire à la réception a un effet dramatique sur les performances. La figure 9.4 présente le débit de transmission que l'on observe pour différentes stratégies d'allocation de la mémoire du côté du récepteur, en fonction de la taille des messages. Ces mesures ont été réalisées sur Myrinet-2000 utilisé par Madeleine 2/BIP, sur des Pentium III 1 GHz sous Linux. La courbe étiquetée `static` correspond à une allocation statique, c'est-à-dire un tampon alloué une fois pour toutes au démarrage du programme et réutilisé d'une réception à l'autre. Elle a une progression normale avec un plafond à 241 Mo/s, soit 96 % du débit nominal du matériel ce qui est un bon résultat. La légère rupture de pente à 8 Ko correspond à un changement de méthode de communication (copie autorisée en-deçà, rendez-vous systématique au-delà) et est donc normale, sans rapport avec l'allocation de la mémoire. Les courbes `malloc` et `new` correspondent respectivement à une allocation dynamique en réception par la primitive `malloc()` de la *libc* et l'opérateur `new` de C++. Nous observons que pour des messages de taille inférieure à 64 Ko, le comportement est similaire à la stratégie `static` avec un surcoût allant jusque 40 μs. Au-delà de 64 Ko, le débit tombe à un pallier autour de 150 Mo/s puis s'effondre pour de très grandes tailles de messages. Ce seuil de 64 Ko correspond à un changement de méthode de l'allocateur `malloc` sous Linux : au-delà de ce seuil, la zone mémoire demandée n'est plus allouée dans un espace commun dédié réutilisé mais est directement demandée au système par la primitive `mmap()` et constitue un

nouveau segment. De ce fait, au-delà du seuil la réception a lieu dans une zone fraîchement allouée par le système d'exploitation et jamais accedée auparavant ; il s'ensuit des défauts de pages en cascade quand on écrit dans cette zone pour la première fois. Les stratégies `malloc` et `new` ont des comportement très similaires, `new` ayant des performances légèrement inférieures à cause de l'initialisation de la zone par l'opérateur `new` lui-même qui introduit un surcoût supplémentaire. La courbe `init` correspond à une réception dans un tampon alloué par `malloc()` puis pré-initialisé avant la réception réseau proprement dite. Sans surprise, les stratégies `new` et `init` ont le même comportement ce qui confirme que la différence entre `malloc` et `new` provient effectivement du surcoût d'initialisation.

Pour vérifier que le comportement problématique au-delà de 64 Ko des stratégies `malloc` et `new` est bien dû au surcoût introduit par les défauts pages lors d'une allocation dynamique, nous proposons une modélisation du débit qui prenne en compte ce temps supplémentaire. Notons T le temps total de transmission d'un volume de données de l octets. Soit D le débit global de cette transmission. Nous avons $D = l/T$. Nous décomposons le temps total en $T = t_{\text{net}} + t_{\text{init}} + t_{\text{malloc}}$ où t_{net} représente le temps de transmission sur le réseau, t_{malloc} le temps d'appel à `malloc`, et t_{init} le temps d'initialisation (défaut de page) pour que la zone allouée puisse effectivement recevoir des données. Nous avons instrumenté le code qui a servi à ces tests pour mesurer t_{malloc} et t_{init}. Nos observations montrent que sur les machines de test, t_{malloc} est modélisable sous forme $t_{\text{malloc}} = \beta_{\text{malloc}} + l\tau_{\text{malloc}}$ avec $\beta = 0.5\,\mu\text{s}$ et $\tau = 0.5\,\mu\text{s/Mo}$. Une approximation de t_{init} est mesurée en instrumentant une version modifiée du test `init`, modification destinée à provoquer les défauts de pages sans le surcoût d'une initialisation complète ; on obtient une valeur $t_{\text{init}}(l)$ en fonction de l qui démarre aux alentours de 1400 Mo/s pour $l < 64$ Ko et tend asymptotiquement vers 360 Mo/s. Le temps total de transmission se décompose donc en $T(l) = t_{\text{net}}(l) + t_{\text{init}}(l) + \beta_{\text{malloc}} + l\tau_{\text{malloc}}$. En l'exprimant sous forme de débit D, nous obtenons alors :

$$\frac{1}{D(l)} = \frac{1}{D_{\text{net}}(l)} + \frac{1}{D_{\text{init}}(l)} + \frac{\beta_{\text{malloc}}}{l} + \tau_{\text{malloc}}$$

La courbe `simulation` de la figure 9.4 correspond à cette formule en prenant pour $D_{\text{net}}(l)$ les valeurs du cas `static` et pour $D_{\text{init}}(l)$ les approximations mesurées en instrumentant le cas `init`. Nous remarquons que la simulation correspond parfaitement à ce que l'on observe pour les stratégies `malloc` et `new` jusqu'à 64 Ko et qu'au-delà on retrouve le même comportement tendant vers 150 Mo/s. La différence observée entre 64 Ko et 512 Ko provient probablement du fait que l'entrelacement entre les défauts de pages et la réception par DMA sur le résaux Myrinet-2000 n'est pas parfait et conduit dans la pratique à un plus faible débit que ce que l'on pourrait attendre.

L'étude de ce modèle confirme que la réception vers des zones mémoire allouées dynamiquement conduit à une dégradation des performances lors de l'utilisation de réseaux très rapides. Cette dégradation n'est pas le fait d'une implémentation médiocre de `malloc` dans la *libc* ; elle est causée intrinsèquement par la réception dans une zone allouée dynamiquement. Dans le cas de Myrinet-2000, la réception vers une zone alloué dynamiquement conduit à un débit de l'ordre de 60 % du débit nominal du matériel, au lieu de 96 % obtenus pour une réception vers une zone allouée statiquement. Cet effet, très visible sur les réseaux Myrinet et Myrinet-2000, a également été constaté dans une moindre mesure sur les réseaux SCI. Le facteur limitant est donc bien la réception depuis le réseau vers une zone mémoire allouée dynamiquement.

Un allocateur semi-statique. Par nature, les réceptions de gros blocs de données par les implémentations CORBA se font dans des zones allouées par l'ORB lui-même. Il réalise cette allocation par `malloc` ou `new`, ce qui au final se ramène à une allocation dynamique. Ce problème est généralisable à tous les exécutifs qui sont basés sur le principe RPC/RMI, avec une réception entièrement gérée par l'exécutif.

Puk offre donc un espace mémoire pré-alloué mis à disposition des exécutifs de type RPC/RMI. Ceux-ci peuvent alors effectuer les réceptions depuis le réseau vers des zones déjà allouées. Cet allocateur est donc qualifié de *semi-statique* : il présente des caractéristiques statiques — l'espace mémoire reste alloué au processus —, et des caractéristiques dynamiques — cet espace est subdivisé en zones allouées dynamiquement aux exécutifs pour leurs réceptions. L'occupation de ces zones de mémoire est temporaire, les executifs ne l'utilisant que lors des appels de méthode à distance, généralement

courts. *Puk* alloue statiquement un ensemble de tampons en mémoire, puis les attribue dynamiquement aux exécutifs avec une stratégie *round robin*. Ce mécanisme n'est utilisé que pour les gros blocs (typiquement, au-delà de 64 ko d'après l'étude précédente). Ce mécanisme est intégré à ces exécutifs sans avoir besoin de les modifier grâce à l'utilisation d'intercepteurs de fonctions qui agissent au moment de l'édition de liens (option `-wrap` de l'éditeur de liens GNU, `-i` sur BSD).

9.2.2 Gestionnaire de tâches : *TaskManager*

Le module *TaskManager* est le module qui gère les processus de PadicoTM. *Puk* dirige les opérations de *bootstrap* ; une fois le cœur complètement chargé, *TaskManager* devient le maître et *Puk* est utilisé comme une bibliothèque utilisée pour les actions sur les modules et les accès mémoire. Le rôle du *TaskManager* est essentiellement de prendre en charge les aspects “système” des processus : accès au multi-threading, gestion des tâches de fond, ordonnancement et synchronisation des opérations sur les modules, gestion de files de requêtes, que nous ne détaillons pas.

Le *TaskManager* met en œuvre la boucle principale d'évènements, qui attend les requêtes — opérations sur les modules, observation de l'état du processus. Les requêtes peuvent venir d'invocations directes depuis l'intérieur du processus, d'interactions avec l'utilisateur sur la console, ou de pilotage à distance par l'intermédiaire d'un module approprié (appelé *gatekeeper*) qui réalise l'interface entre un client et le *TaskManager*. Le *TaskManager* contient par défaut des *gatekeepers* pour pilotage à distance *via* CORBA, SOAP, et une variante de XML-RPC.

9.2.3 Accès arbitré au réseau : *NetAccess*

La couche d'accès arbitré au réseau est implémentée par le module *NetAccess*, subdivisé en trois parties : *SysIO* pour les entrées/sorties systèmes, *MadIO* pour les communications par Madeleine, et une partie principale pour les interactions.

9.2.3.1 *NetAccess SysIO*

NetAccess SysIO est la partie de *NetAccess* dédiée à l'arbitrage pour l'accès aux entrées/sorties systèmes ; les entrées/sorties systèmes sont celles qui se programment à l'aide de descripteurs de fichiers, c'est-à-dire le réseau par *sockets* mais aussi les fichiers et entrées/sorties standard. Lorsque l'on utilise la bibliothèque de *multi-threading* Marcel, selon la classification de la page 107, les entrées/sorties systèmes sont multiplexées, mais sans-réentrance native. *SysIO* met donc en œuvre un scrutateur permanent pour gérer les réception et les émissions qui sont gérées par des fonctions de rappel et sérialisées pour éviter les problèmes de réentrance.

L'interface de programmation de *SysIO* est donnée à la figure 9.5. Elle est organisée autour de traitants associés aux descripteurs de fichiers. Lorsqu'un module désire réaliser des entrées/sorties systèmes, dès qu'il a obtenu un descripteur de fichier (*fd*), il l'enregistre dans le scrutateur de *SysIO* à l'aide de `sysio_register` et lui associe un traitant. Il peut ensuite masquer l'appel du traitant en fonction des évènements à l'aide de `sysio_activate` et `sysio_deactivate` et des constantes `SYSIO_EVENT_*`. Le scrutateur appelle le traitant à chaque fois que le descripteur est détecté comme pouvant être écrit, lu ou en erreur, et que l'évènement correspondant est activé dans le masque. L'opération proprement dite peut alors être réalisée dans le traitant.

Quand un adaptateur est à l'initiative d'une entrée/sortie sur *SysIO*, pour améliorer la réactivité, il peut demander de forcer une scrutation par la primitive `sysio_refresh` pour invoquer immédiatement les traitants prêts. Ceci supprime le surcoût de latence introduit s'il fallait attendre la prochaine itération normale du scrutateur. Toutefois, il faut prendre garde aux abus possibles par un rythme de scrutations forcées soutenu. Si un adaptateur entre en attente active sous l'effet d'un exécutif qui effectue des attentes actives — MICO ou Kaffe par exemple —, alors le scrutateur mobilise une trop grande part des ressources pour un résultat à l'efficacité discutable. Comme proposé au chapitre 8, *SysIO* met en œuvre un contrôle des scrutations abusives sous forme d'un *quota* : le nombre de scrutation forcées entre deux scrutations normales est mémorisé ; au-delà d'un seuil, le forçage n'est pas effectué et est

```
#define SYSIO_EVENT_READ   0x01
#define SYSIO_EVENT_WRITE  0x02
#define SYSIO_EVENT_ERROR  0x04
typedef int (*sysio_callback_t)(int fd, unsigned int what, void*key);
      sysio_t sysio_register   (int fd, sysio_callback_t cb, void*key);
         void sysio_activate   (sysio_t io, unsigned int what);
         void sysio_deactivate (sysio_t io, unsigned int what);
         void sysio_unregister (sysio_t io);
          int sysio_status     (sysio_t io, unsigned int what);
         void sysio_refresh    (sysio_t io);
```

FIG. 9.5 – *L'interface de* SysIO*.*

remplacé par une attente de régulation pour modérer l'agressivité de l'exécutif appelant. Une partie du décompte est reportée d'une itération à l'autre pour lisser l'action de la régulation dans le temps.

Scrutateur *SysIO* avec Marcel-mono. Avec la version Marcel-mono, il n'est pas possible de réaliser une scrutation bloquante sans bloquer tout le processus. Les deux possibilités sont donc une scrutation non-bloquante avec régulation (stratégie 2, avec les stratégies définies à la section 8.4) ou une scrutation par l'ordonnanceur Marcel lui-même (stratégie 3).

En scrutation non-bloquante avec régulation, notre étude de l'équité a mis en avant que ce cas amène la problématique du *délai d'attente* pour réguler la compétition et éviter une scrutation trop agressive. *SysIO* utilise la méthode adaptative présentée basée sur les modes rafale/repos pour préserver de bonnes performances sans pénaliser les autres *threads*. Dans ce cas, le scrutateur est composé d'une boucle qui consulte à l'aide d'un `select` ou `poll` non-bloquant l'état de tous les descripteurs enregistrés. Il appelle ensuite les uns après les autres les traitants des descripteurs prêts et activés dans le masque, et effectue ensuite une attente de régulation de l'équité. Cette attente obéit à un mode rafale/repos : si aucune action (traitant appelé, descripteur activé, etc.) n'a été détectée pendant les dernières itérations (dont le nombre est paramétrable), alors le scrutateur passe en mode *repos*, avec une attente longue, de façon à ne pas concurrencer les autres *threads* qui sont probablement dans des calculs. Dès qu'une activité est détectée, le scrutateur se réveille et passe en mode *rafale*, avec un simple `yield` à la place de l'attente de régulation, voire une très courte attente active.

En scrutation par l'ordonnanceur, la scrutation à l'aide de `select` ou `poll` est directement appelée dans un traitant par l'ordonnanceur Marcel, puis envoie un signal au *thread* de *SysIO* qui utilise les mêmes mécanismes d'invocation des traitants des descripteurs prêts que la stratégie précédente.

Scrutateur *SysIO* avec Marcel-SMP. Avec la version Marcel-SMP, il est possible de réaliser une scrutation bloquante (stratégie 1) et une certaine réentrance est assurée, à condition que les opérations concurrentes soient dans des *threads* noyau différents. La possibilité de scrutation bloquante règle la question de l'équité en supprimant l'attente active au profit d'une attente gérée par le noyau ; ceci supprime le besoin du délai de régulation. *SysIO* maintient deux ensemble de descripteurs de fichiers : un ensemble de descripteurs à tester — sur lesquels l'attente est potentiellement indéfinie —, et un ensemble de descripteurs qu'il est inutile de tester car leur masque est désactivé. Optionnellement, les traitant peuvent être invoqués par plusieurs *threads* noyau pilotés par le scrutateur. Ceci introduit un plus haut niveau de concurrence appréciable sur les machines SMP, mais ajoute un changement de contexte ce qui dégrade légèrement la latence.

Scrutateur *SysIO* avec activations. L'ordonnanceur de *threads* Marcel peut utiliser des mécanismes appelés "*activations*". Les activations [64] offrent un dialogue entre le système d'exploitation et l'ordonnanceur de *threads* lors des appels système bloquants. Plutôt que de bloquer le processus lors d'un tel appel bloquant, le noyau informe l'ordonnanceur de *threads* qui peut alors par exemple donner la main à un autre *thread*. L'utilisation de ces mécanismes ne modifie pas l'interface de Marcel. Ils peuvent donc être utilisés de façon transparente par PadicoTM.

9.2.3.2 *NetAccess MadIO*

NetAccess MadIO est la partie de *NetAccess* dédiée à l'arbitrage pour l'accès à la bibliothèque de communication Madeleine. Les communications Madeleine se font sur des *canaux* de communication, un canal étant un espace de communication recouvrant l'ensemble des nœuds d'une grappe Madeleine. Le nombre de canaux distincts fournis par Madeleine dépend des capacités du matériel. Ce nombre est borné, avec une borne relativement basse — typiquement 1, 2, 4 ou 6, sur les réseaux SCI et Myrinet, selon le type de réseau et de pilote. Cette contrainte provient du fait que Madeleine n'offre que le multiplexage réalisable par le pilote. Nous appelons dans PadicoTM ces canaux de communication de Madeleine des *canaux physiques*, qui sont décrits par des structures PadicoTM de type `padico_phys_channel_t`.

Selon notre classification établie en page 107, Madeleine est de type à n canaux natifs, avec scrutation non-bloquante ou bloquante à attente passive. *MadIO* met donc en œuvre un multiplexage quasi-illimité, et n scrutateurs permanents. Étant donnée les problèmes de compétition qui apparaissent avec les scrutations non-bloquantes, nous choisissons la scrutation bloquante. L'arbitrage concerne essentiellement l'accès aux canaux : gestion des canaux physiques, multiplexage en canaux logiques, et affectation des canaux logiques sur les canaux physiques. L'interface de *MadIO* est donc très similaire à celle de Madeleine, à l'exception près que la réception se fait par messages actifs. Les échanges de données ont la forme de *messages*, eux mêmes découpés en *paquets*. L'interface de *MadIO* est donnée à la figure 9.6 — les préfixes en `padico_` y sont volontairement omis pour une meilleure lisibilité.

Multiplexage sur Madeleine. Au-dessus des canaux physiques, *MadIO* offre des canaux logiques en nombre virtuellement illimité. Ces canaux sont multiplexés à l'aide d'une clef sur 32 bits, ce qui autorise potentiellement plus de 4 milliards de canaux. Ces canaux multiplexés sont décrits par des structures de type `padico_xchn_t`. Lors d'une communication, un canal multiplexé est affecté à un canal physique. L'affectation est dynamique pour utiliser efficacement le niveau de concurrence offert par Madeleine, en accord avec les principes énoncés en section 8.4 pour les réseaux à n canaux. En pratique, n'importe quel canal libre convient à condition qu'aucune autre opération sur la même connexion ne soit en cours sur un autre canal, ceci pour respecter l'ordre des messages sur chaque connexion.

Il se pose alors le problème de l'attribution d'une clef de multiplexage qui identifie un canal multiplexé de façon unique sur tout un canal physique. *MadIO* partitionne l'espace de ces clefs en deux : une moitié où les clefs sont attribuées dynamiquement, et une moitié à attribution implicite. La partie à attribution implicite correspond à un ensemble de canaux implicitement ouverts, dont la valeur de la clef est connue de tous les nœuds par convention. Il suffit d'invoquer la fontion `padico_xchn_implicit_init` pour déclarer le traitant et récupérer un descripteur. Leur usage est dangereux car lorsqu'on envoie sur un canal implicitement ouvert, il n'y a aucun moyen de savoir qu'un traitant de messages actifs est enregistré chez le récepteur. Leur usage est donc réservé aux cas où l'on *sait* que tous les nœuds peuvent recevoir : les canaux dédiés au contrôle des canaux dynamiques et ceux utilisés par le sélecteur.

Les autre canaux sont dynamiques. Ils peuvent être ouverts et modifiés à tout moment, et peuvent également être partiels, c'est-à-dire couvrir un sous-ensemble des nœuds. Les canaux dynamiques sont ouverts à l'initiative d'un nœud à l'aide de la primitive `padico_xchn_dyn_create`, qui les diffuse à tous les autres nœuds. Ce nœud est alors le *leader* du canal. Les autres nœuds peuvent ensuite déclarer leur appartenance au canal à l'aide de la primitive `padico_xchn_dyn_join`; ceci envoit une notification à tous les membres, via le *leader*, que le nœud est prêt à recevoir. Si la déclaration d'appartenance est faite sur un canal qui n'existe pas encore, la primite attend que le canal "apparaisse", créé par un autre nœud. Puisqu'on ne connaît pas *a priori* la clef d'un canal dynamique, on l'identifie à l'aide d'une chaîne de caractère. Pour éviter toute collision de clef, l'espace des clefs dynamiques et partitionné en sous-espaces distincts en fonction du numéro du *leader*.

Agrégation d'en-têtes. Le multiplexage demande l'adjonction de la clef de 32 bits à chaque message échangé. Pour éviter le surcoût de l'envoi d'un paquet supplémentaire ou d'une copie de toutes les

```
typedef void (*xchn_callback_t)(xchn_cnx_t cnx, void*key,
                             void*hdr, size_t hdrsize);
        xchn_t xchn_implicit_init(const char* xchn_id, uint32_t tag,
                             xchn_callback_t handler, void*key);
        xchn_t xchn_dyn_create(const char* xchn_id,
                             xchn_callback_t handler, void*key);
        xchn_t xchn_dyn_join(const char* xchn_id,
                             xchn_callback_t handler, void*key);
          void xchn_close(xchn_t xchn);
    xchn_cnx_t xchn_begin_packing(xchn_t xchn, int dest_node,
                             void**hdr, size_t*hdrsize);
          void xchn_pack(xchn_cnx_t cnx, void*ptr, size_t size,
                             send_mode_t smode, recv_mode_t rmode);
          void xchn_end_packing(xchn_cnx_t cnx);
          void xchn_unpack(xchn_cnx_t cnx, void*ptr, size_t size,
                             send_mode_t smode, recv_mode_t rmode);
          void xchn_end_unpacking(xchn_cnx_t cnx);
```

FIG. 9.6 – *L'interface de* MadIO.

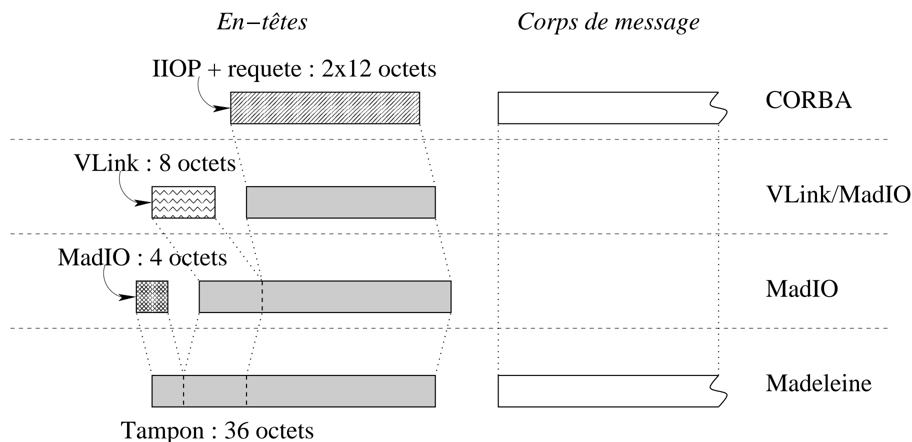

FIG. 9.7 – *Agrégation d'en-têtes : exemple du cheminement d'un message* CORBA *transporté par Madeleine.*

données — deux solutions qui ne sont pas satisfaisantes du point de vue des performances, aspect crucial sur les réseaux gérés par *MadIO* —, le modèle propose d'agréger les en-têtes. En effet, les adaptateurs et les exécutifs utilisent pour la plupart des messages composés d'un en-tête et d'un corps, l'en-tête étant indispensable ne serait-ce que pour indiquer la longueur du corps de message. *MadIO* met pour cela à disposition des couches supérieures un tampon destiné à construire l'en-tête agrégé. La construction incrémentale de l'en-tête se fait alors de bas en haut. À l'initialisation d'un message, *MadIO* alloue un tampon d'une taille déterminée ; chaque couche écrit sa partie d'en-tête dans le tampon et passe un pointeur sur la partie restante à la couche supérieure. Le reste du message a une forme entièrement libre et peut contenir plusieurs paquets.

La figure 9.7 illustre un exemple typique de CORBA au-dessus de Madeleine. *MadIO* alloue un tampon de 36 octets destiné à être envoyé en tant qu'en-tête, et y place ses 4 octets. L'adaptateur *VLink/MadIO* ajoute ses 8 octets d'en-tête ; la personnalité *socket* (non représentée sur la figure) n'ajoute aucun en-tête. L'ORB place un en-tête de 24 octets (12 octets d'en-tête IIOP, 12 octets d'information de requête). Le corps de message n'est pas modifié en traversant l'empilement des couches ; il est transmis sans copie. Si l'en-tête alloué par *MadIO* n'est pas rempli au moment de l'envoi, le code utilisateur peut choisir de placer le début du corps de message dans l'espace restant ; *VLink* procède ainsi par exemple. Ceci permet d'économiser un paquet dans le cas où l'espace restant dans l'en-tête agrégé peut contenir toutes les données.

Réception de messages actifs. Puisque Madeleine est du type à n canaux, selon la classification donnée à la section 8.4, *MadIO* met en œuvre un scrutateur — un *thread* de réception — pour chaque canal physique. Chaque scrutateur passe l'essentiel de son temps en attente de réception sur le canal dont il est responsable. Quand un message arrive, le thread reçoit l'en-tête agrégé, résout la clef de multiplexage de l'en-tête en un descripteur de canal multiplexé, et invoque le traitant de *messages actifs* correspondant au canal multiplexé enregistré à l'initialisation. Le traitant déballe la partie de l'en-tête le concernant et procède à la réception du corps de message à l'aide des primitives `xchn_unpack`, avant de rendre la main au thread de scrutation. Pour éviter les interblocages, un traitant ne doit pas réaliser d'opérations à attente indéterminée, telle que : envoi sur le réseau, attente sur condition, ou certaines prises de verrous. Pour n canaux physiques, ce dispositif permet de traiter de façon concurrent n messages entrants.

Émission. Lorsqu'une émission est demandée, *MadIO* vérifie si le destinataire a activé un traitant sur le canal demandé, puis cherche un canal physique pour transporter le message. N'importe quel canal physique libre (*ie.* sans émission en cours) convient, à condition qu'il ne soit pas réservé par un adaptateur prioritaire qui se réserve le droit d'émettre à tout moment sur son canal réservé. L'émission est ensuite gérée d'un façon similaire à Madeleine, à l'exception des en-têtes agrégés.

9.2.3.3 Stratégies d'entrelacement

La compétition entre *SysIO* et *MadIO* est gérée en réglant l'agressivité relative des scrutateurs. En réalité, la scrutation *MadIO* n'a pas besoin de réglage particulier car la scrutation Madeleine est intégrée à l'ordonnanceur de *threads* Marcel ; les *threads* de *MadIO* sont simplement placés en classe de priorité "temps réel" offerte par Marcel, de façon à leur donner la main en priorité quand des données doivent être traitées par Madeleine.

L'essentiel du réglage se fait donc dans *SysIO*. Le réglage de l'agressivité de scrutation de *SysIO* est contrôlé par des paramètres modifiables dynamiquement par l'utilisateur. Ces paramètres sont ceux décrits dans le modèle à la section 8.4. Il sont traduit dans *SysIO* en les paramètres suivants :

- `interleave_factor` — ce paramètre contrôle l'agressivité de la boucle de scrutation lorsqu'elle est au repos. Sa valeur représente le temps de régulation attendu explicitement après chaque scrutation. Les valeurs possibles sont : un temps non-nul, zéro (signifiant un `yield` sans délai), -1 indiquant de garder la main.
- `idle_factor` — ce paramètre indique le temps pendant lequel *SysIO* reste en mode *rafale* après la dernière activitée détectée.
- `forced_polling_quota` — ce paramètre contrôle le nombre de `sysio_refresh` autorisés sur un descripteur donné avant d'appliquer la pénalité pour scrutation abusive. Il est donnée en nombre d'appels "abusifs" (forçage de scrutation alors qu'aucune donnée n'est présente) à cette fonction entre deux itérations "normales" du scrutateur.

Ces paramètres permettent un contrôle fin de l'agressivité de *SysIO*, et donc de l'entrelacement entre *SysIO* et *MadIO*— plus *SysIO* prend la main souvent, moins *MadIO* est réactif. Ces paramètres n'ont un sens que lorsque la stratégie de scrutation utilise une scrutation non-bloquante et une régulation (stratégie 2) ; ces paramètres, qui servent à régler la fréquence de scrutation, ne concernent pas la version de *SysIO* qui utilise `marcel_poll`.

SysIO propose de plus des paramètres pour régler l'agressivité des opérations elles-mêmes, une fois que l'arbitre leur a donné la main. Ces paramètres permettent de contrôler diverses stratégies :

- `strategy_preempt` — ce paramètre contrôle la désactivation ou non de la préemption de l'ordonnanceur de Marcel pendant qu'une opération d'entrée/sortie longue a lieu.
- `strategy_retry` — ce paramètre contrôle la stratégie de la deuxième chance. Quand un appel système d'entrée/sortie n'a pas traité en totalité les données, cette stratégie réessaie immédiatement et pendant une période déterminée de terminer l'opération sans rendre la main. Cette stratégie est particulièrement utile pour TCP/IP sur Ethernet, qui émet paquet par paquet.

- `retry_factor` — ce paramètre détermine le temps alloué à la stratégie de la deuxième chance avant d'abandonner l'opération et de la considérer comme terminée même si toutes les données ne sont pas traitées. Ce paramètre est donné sous la forme d'un débit nominal, de façon à déterminer le temps au-delà duquel il n'est pas normal que l'opération ne soit pas terminée.

Typiquement, nous prenons comme paramètres par défaut : `interleave_factor = 1` ms, `idle_factor = 20` ms, `forced_polling_quota = 5`, `strategy_preempt=true`, `strategy_retry=true`, `retry_factor = 10` Mo/s. Cette configuration se veut *neutre*, c'est-à-dire qu'elle ne désavantage particulièrement ni *MadIO*, ni *SysIO*, ni les *threads* applicatifs.

9.2.3.4 Discussion

NetAccess fournit une base solide pour construire diverses abstractions de communication : les services offerts par *NetAccess* sont un accès entièrement réentrant et multiplexé aux réseaux haute performance et aux réseaux IP. De part son rôle — gérer finement les interactions et multiplexer à bas niveau —, *NetAccess* est susceptible d'être modifié si l'on désire utiliser un réseau supplémentaire ou changer de méthode d'accès à un réseau. Cependant, les modifications à apporter sont locales. Par exemple, pour remplacer Madeleine par une autre bibliothèque d'accès aux réseaux haute performance, il suffit d'implémenter l'API de la figure 9.6 sur cette nouvelle bibliothèque. Les bibliothèques de communication pour réseaux du parallélisme ayant des interfaces relativement semblables, la tâche ne devrait pas être trop ardue.

9.3 Le niveau abstrait dans PadicoTM

Cette section est consacrée au niveau abstrait dans PadicoTM, à savoir la définition des interfaces abstraites : *VLink* en réparti, et *Circuit* en parallélisme à mémoire distribuée, et à la sélection et l'assemblage d'adaptateurs. Les adaptateurs proprement dit sont décrits dans la section suivante.

9.3.1 L'abstraction pour le réparti : *VLink*

L'abstraction du réparti dans PadicoTM est appelée *VLink* – abréviation de *Virtual Link* (lien virtuel) ; ce sont des liens abstraits découplés de la notion de lien physique. L'interface *VLink* est détaillée à la figure 9.8. Cette interface est présentée par des adaptateurs qui sont des composants ; pour les rendre composables par une tierce partie, l'interface est en réalité une structure contenant des pointeurs vers des fonctions. Nous les présentons ici sous forme de prototypes de fonctions pour alléger les notations.

Les liens sont manipulés par des descripteurs de type `vlink_t`. Ce sont des liens point-à-point, entre deux nœuds, topologie de base du réparti. L'adressage est réalisé par les structures standard `struct sockaddr` de l'en-tête C `<netinet/in.h>` ; ces adresses sont composées d'un champ qui indique la nature de l'adresse (par exemple `AF_INET` pour *internet*) et d'une partie variable dépendant du type d'adresse. Cette structure est donc capable d'exprimer un large éventail de types d'adresses et présente l'avantage d'être standard. Les adaptateurs peuvent comprendre leur propres classes d'adressage ; tous les adaptateurs de type réparti doivent cependant être en mesure de manipuler les adresses de la classe `AF_INET`. Les adaptateurs qui proposent des communications *locales* doivent être en mesure de comprendre des adresses `AF_UNIX`.

L'interface *VLink* est construite selon les mécanismes asynchrones présentés en section 6.2 : un ensemble de primitives est utilisé pour lancer les opérations sans attendre leur terminaison, la synchronisation et la scrutation étant assurées par un *observateur*. Les observateurs ont le type `vobs_t` et manipulent des attributs dont les étiquettes sont énumérées par `vobs_attr_t`. Les primitives sont organisées en quatre ensembles : la fabrique, la gestion de lien, l'échange de messages, et l'observateur.

```
/* fabrique */
typedef struct vlink_s* vlink_t;
  vlink_t vlink_create  (void)
     void vlink_destroy (vlink_t l);
/* gestion de lien */
      int vlink_bind    (vlink_t l, const struct sockaddr*, socklen_t);
      int vlink_listen  (vlink_t l, int backlog);
      int vlink_connect (vlink_t l, const struct sockaddr*, socklen_t);
      int vlink_shutdown(vlink_t l, int how);
/* échange de messages */
  ssize_t vlink_send    (vlink_t l, const void*ptr, size_t size);
  ssize_t vlink_recv    (vlink_t l, void*ptr, size_t);
/* observateur */
typedef enum { status, error, observers, binding, listener, remote, wbox, rbox }
          vobs_attr_t;
   vobs_t vobs_of_vlink (vlink_t);
    void* vobs_acquire  (vobs_t o, vlink_attr_t attr);
    void  vobs_release  (vobs_t o, vlink_attr_t attr);
    void  vobs_wait     (vobs_t o, vlink_attr_t attr);
```

FIG. 9.8 – *L'interface abstraite* VLink.

```
int rc;
struct rbox_attr_s* rbox;
rbox = (struct rbox_attr_s*)
  vobs_acquire(vobs, rbox_attr);
rc = rbox->read_done;
vobs_release(vobs, rbox_attr);
```

```
struct rbox_attr_s* rbox;
rbox = (struct rbox_attr_s*)
  vobs_acquire(vobs, rbox_attr);
while (rbox->read_done < len)
  vobs_wait(vobs, rbox_attr);
vobs_release(vobs, rbox_attr);
```

FIG. 9.9 – *Exemples d'utilisation de l'observateur : à gauche, récupérer l'état d'avancement d'une réception dans la variable* `rc` *; à droite, attendre que la réception postée ait reçu au moins* `len` *octets.*

L'observateur. Un observateur permet la scrutation de l'état d'un lien, décrit à l'aide d'attributs. Les primitives d'interaction avec un observateur sont :

`vobs_acquire` — cette primitive demande à l'observateur de fournir une référence sur l'attribut spécifié, retourné dans un pointeur `void*`, dont l'interprétation dépend de l'attribut ; par convention, dans *VLink* les attributs d'étiquette *xxx* ont pour type `struct` *xxx*`_attr_s`. L'appelant acquiert ainsi l'accès exclusif en lecture/écriture à cet attribut jusqu'à ce qu'il le relâche. Par convention, on renvoit `NULL` si l'attribut n'existe pas sur le lien considéré.

`vobs_release` — cette primitive relâche l'accès à un attribut, et met à jour les éventuelles modifications qui ont été faites depuis l'acquisition de l'attribut.

`vobs_wait` — cette primitive attend qu'un signal du type spécifié soit généré sur l'observateur.

Un exemple d'utilisation des observateurs est donné à la figure 9.9.

De plus, à chaque fois qu'il reçoit un signal, l'observateur invoque une liste de fonctions de rappel qui ont été préalablement enregistrées. La gestion de la liste de ces fonctions de rappel est réalisée en appliquant la scrutation *via* l'observateur à lui-même. Les fonctions de rappel sont stockées dans un attribut d'étiquette "`observer`" et de contenu `observer_attr_s` correspondant défini par :

```
struct vobs_handler_s {
  void(*callback)(vlink_t l,vlink_attr_t attr, void*);
  void* key;
};
struct observer_attr_s {
  vobs_handler_list_t handlers[MAX_ATTR];
};
```

où `vobs_handler_list_t` représente une liste de `struct vobs_handler_s`. L'enregistrement et la suppression des fonctions de rappel déclenchées par les signaux se fait ainsi en modifiant la liste `handlers`.

```
struct remote_attr_s* remote;
vlink_connect(l, addr, addrlen);
remote = (struct remote_attr_s*)
  vobs_acquire(vobs, remote_attr);
while (!remote->established)
  vobs_wait(vobs, remote_attr);

vobs_release(vobs, remote_attr);
```

```
struct listener_attr_s* listener;
vlink_listen(l, 1);
listener = (struct listener_attr_s*)
  vobs_acquire(vobs, listener_attr);
while(queue_empty(listener->queue))
  vobs_wait(vobs, rbox_attr);
newlink = pop_front(listener->queue);
vobs_release(vobs, rbox_attr);
```

FIG. 9.10 – *Exemple simplifié d'utilisation des primitives* VLink *de gestion de lien : à gauche, un client ; à droite, un serveur.*

Il est clair que les implémentations des primitives de l'observateur dans les différents adaptateurs présentent de nombreuses similitudes d'un adaptateur à l'autre. Dans tous les cas où il est possible d'offrir un accès direct aux structures internes, les fonctions `vobs_acquire` et `vobs_release` se résument à une acquisition et relâchement de verrou et calcul de l'*offset* de l'attribut dans la structure. De même, l'utilisation d'une variable conditionnelle issue de la bibliothèque de *multi-threading* pour assurer `vobs_wait` semble immédiat. Pour implémenter ces mécanismes communs réutilisables en grande partie d'un adaptateur à l'autre, PadicoTM fournit un canevas d'observateur *générique* qui peut servir d'implémentation par défaut et dont il est possible de dériver des variantes.

Un adaptateur ne présentant que les primitives de fabrique et de l'observateur permet de construire des observateurs secondaires dédiés à l'agrégation de plusieurs scrutations, à la façon de `select()`. À sa création, un observateur secondaire initialise ses attributs en fonction de ceux qu'il observe. Pour suivre leur évolution, il enregistre dans chacun des liens observés une fonction de rappel qui envoie un signal à l'observateur secondaire. Puisqu'il ne correspond pas à un seul lien mais à plusieurs liens, les attributs d'un observateur secondaire sont des vecteurs d'attributs simples.

Primitives de gestion de lien et d'échange de messages. La gestion de lien est basée sur les quatre opérations `vlink_bind`, `vlink_listen`, `vlink_connect` et `vlink_shutdown`, et les trois attributs `binding`, `listener` et `remote`. Les primitives sont une transposition directes du modèle de la section 6.2. Les attributs contiennent les informations suivantes :

`binding` contient l'adresse locale du lien. Cette adresse peut avoir été spécifiée par un appel à `vlink_bind` ou être affectée implicitement lors d'un appel à `vlink_connect`. C'est un attribut en lecture seul ; le modifier ne modifie pas le comportement futur de l'adaptateur.

`listener` contient les connexions entrantes en attente sur un descripteur faisant office de serveur. Les actions habituelles sont de tester la longueur de la liste et de retirer le premier élément. Toute autre modification de la liste est permise, mais n'a probablement pas grand sens.

`remote` contient les informations sur le pair distant, à savoir son adresse et un indicateur d'établissement de connexion.

La figure 9.10 illustre l'utilisation de ces primitives pour réaliser un client et un serveur, chacun attendant de façon synchrone la terminaison. Cet exemple est volontairement simplifié et ne gère pas correctement les erreurs.

L'envoi et la réception de données sont basés sur les primitives `vlink_send` et `vlink_recv`, et des boîtes d'émission et réception hébergées respectivement par les attributs `wbox` et `rbox`. Chacune des boîtes est décrite par les informations suivantes : pointeur sur la zone de données, taille totale, taille déjà traitée, interruptibilité de l'opération. La figure 9.11 illustre l'utilisation de ces primitives et attributs pour un envoi de données. L'exemple ne contient volontairement aucune gestion des erreurs. Dans l'exemple, l'émetteur attend l'émission complète avant de continuer. Le récepteur arrête la réception au premier fragment reçu.

Adaptateur d'accès aux fichiers : *VFile*. Toutes les opérations systèmes potentiellement bloquantes sont gérées par *SysIO*, y compris les accès aux fichiers. À part la construction, les fichiers et les connexions réseaux s'utilisent de façon très similaire par l'intermédiaire d'un descripteur. L'accès aux

```
struct wbox_attr_s* wbox;
vlink_send(l, buffer, size);
wbox = (struct wbox_attr_s*)
  vobs_acquire(vobs, wbox_attr);
while (wbox->write_done<size)
  vobs_wait(vobs, wbox_attr);

vobs_release(vobs, wbox_attr);
```

```
struct rbox_attr_s* rbox;
vlink_recv(l, buffer, size);
rbox = (struct rbox_attr_s*)
  vobs_acquire(vobs, rbox_attr);
while(rbox->read_done == 0))
  vobs_wait(vobs, wbox_attr);
rbox->read_posted = 0;
vobs_release(vobs, rbox_attr);
```

FIG. 9.11 – *Exemple simplifié d'utilisation des primitives* VLink *d'échange de données : à gauche, un envoi ; à droite, une réception. Le récepteur s'arrête au premier fragment reçu.*

fichiers au niveau abstrait est réalisé par une variante de *VLink* appelée *VFile*. L'interface *VFile* inclut les primitives de fabrique, de lecture écriture, et l'observateur de *VLink* ; les primitives de gestion de lien sont remplacée par les primitives de gestion de fichiers : `open`, `creat`, `lseek`, `fstat` et `mmap`, chacune étant une version abstraite de l'appel système correspondant. PadicoTM comprend un adaptateur *VFile/SysIO* qui offre un accès aux fichiers réels. Il est envisageable d'implémenter d'autres adaptateurs pour, par exemple, réaliser des accès aux fichiers à distance ou un système de fichiers parallèle.

Discussion. L'interface *VLink* présentée permet d'implémenter des accès aussi bien synchrones qu'asynchrones. Elle est adaptée pour être utilisée par diverses personnalités. Le code pour l'utiliser et relativement compact par rapport à d'autres interfaces asynchrones telles que Posix AIO. Le mécanismes des observateurs est conçu pour que la majorité des implémentations utilise la version par défaut qui autorise des interactions particulièrement optimisées.

9.3.2 L'abstraction pour le parallélisme à mémoire distribuée : *Circuit*

L'interface abstraite pour le parallélisme à mémoire distribuée est appelée *Circuit*. Un *Circuit* est un canal abstrait de communication reliant un ensemble de nœud. L'interface *Circuit* est très similaire à l'interface de *MadIO*, à ceci près que la topologie d'un *Circuit* est indépendante de la topologie physique.

9.3.2.1 Interface abstraite *Circuit*

La figure 9.12 présente l'interface *Circuit*. Tout comme les *VLink*, cette interface est en réalité une structure contenant des pointeurs vers des fonctions pour la composabilité par une tierce partie. Nous les présentons cependant sous forme de prototypes de fonctions pour alléger les notations. Ces primitives sont organisées en trois catégories, suivant le modèle proposé en section 6.3 : administration de circuit, échange de messages, et opérations collectives. La construction et la destruction d'un circuit sont réalisés par les primitives suivantes :

`circuit_create` — cette primitive a pour but de construire un nouveau circuit basé sur le groupe de nœuds passé en paramètre. Pour que tous les nœuds utilisent le même ensemble de nœuds pour le circuit, un circuit se base sur un *groupe* déjà créé dont tout les membres sont d'accord sur son contenu (voir ci-dessous la description des *groupes*). La création d'un circuit est une opération bloquante du point de vue de l'appelant ; chaque nœud doit appeler la primitive. L'appel retourne uniquement quand tous les nœuds ont réalisé l'initialisation du circuit.

`circuit_destroy` — cette primitive attend que tous les nœuds entrent en phase de terminaison, puis détruit le circuit. Comme pour la création, tous les nœuds doivent invoquer la destruction d'un circuit pour que l'opération puisse avoir lieu.

L'envoi est basé sur des messages structurés en paquets. Les messages sont délimités par des appels aux primitives `begin_packing` et `end_packing` ; chaque paquet est empaqueté par la primitive `pack`. En réception, un début de message est signifié par l'invocation de la fonction traitante associée

```
 typedef void (*circuit_callback_t)(circuit_cnx_t cnx, void*key);
    circuit_t circuit_create(char* circuit_id, char* group_id,
                             circuit_callback_t handler, void*key);
         void circuit_destroy(circuit_t c);
          int circuit_size   (circuit_t c);
          int circuit_rank   (circuit_t c);
circuit_cnx_t circuit_begin_packing(circuit_t c, int dest_node);
         void circuit_pack(circuit_cnx_t cnx, void*ptr, size_t size,
                             send_mode_t smode, recv_mode_t rmode);
         void circuit_end_packing(circuit_cnx_t cnx);
         void circuit_unpack(circuit_cnx_t cnx, void*ptr, size_t size,
                               send_mode_t smode, recv_mode_t rmode);
         void circuit_end_unpacking(circuit_cnx_t cnx);
```

FIG. 9.12 – *L'interface abstraite* Circuit.

```
msg = circuit_begin_packing(...);
circuit_pack(msg, &size, sizeof(int),
 send_CHEAPER, receive_EXPRESS);

circuit_pack(msg, array, size,
 send_CHEAPER, receive_CHEAPER);
circuit_end_packing(msg);
```

```
void msg_handler(circuit_cnx_t msg,...) {
 circuit_unpack(msg, &size, sizeof(int),
   send_CHEAPER, receive_EXPRESS);
 array = malloc(size);
 circuit_unpack(msg, array, size,
   send_CHEAPER, receive_CHEAPER);
 circuit_end_unpacking(conn);
}
```

FIG. 9.13 – *Exemple d'échange d'un message de longueur arbitraire avec l'interface* Circuit.

au circuit ; chaque paquet est extrait par un appel à la primitive `unpack`, la fin de l'extraction du message étant signalée par un appel à `end_unpacking`. Les primitives correspondantes sont :

`circuit_begin_packing` — Débute un nouveau message

`circuit_pack` — Insère un paquet dans un message

`circuit_end_packing` — Termine une émission de message

`circuit_unpack` — Extrait un paquet d'un message

`circuit_end_unpacking` — Termine une réception de message

Il n'y a pas de `begin_unpacking` ; le début d'un message est implicite lors de l'invocation du traitant de réception. Les envois et réceptions sont accompagnés d'indications sur la méthode à employer. À chaque paquet est associé un mode d'émission et un mode de réception. Les modes disponibles en envoi sont les suivants :

`send_SAFER` — La zone de mémoire du bloc de données peut être réutilisée par l'application tout de suite après `pack` sans affecter les données envoyées ;

`send_LATER` — L'émission réelle du paquet est différée de façon à prendre en compte la valeur de la zone mémoire au moment de l'appel à `end_packing` ;

`send_CHEAPER` — La plate-forme est libre d'ordonnancer l'envoi de ce paquet au meilleur moment pour les performances. Aucune copie n'étant faite, le bloc de données ne doit pas être modifié avant l'appel `end_packing`.

Les modes suivants sont disponibles en réception :

`receive_EXPRESS` — Les données sont disponibles immédiatement après le `unpack` ;

`receive_CHEAPER` — Aucune garantie n'est donnée sur la disponibilité des données avant l'appel à `end_unpacking`.

La figure 9.13 illustre l'envoi d'une chaîne de longueur arbitraire en utilisant ces primitives (en pseudo-code).

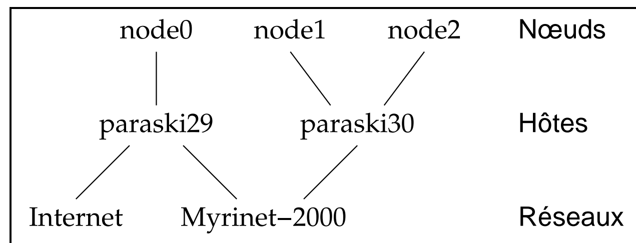

FIG. 9.14 – *Exemple de topologie d'étude : trois nœuds répartis sur deux machines, une seule machine connectée à internet.*

9.3.2.2 Le pilote **multi** et la gestion des groupes

Un *groupe* est une liste ordonnée de nœuds PadicoTM. Un groupe est décrit par une structure de type `group_t` et identifié par un nom symbolique sous forme de chaîne de caractères. Un groupe est créé localement sur un nœud puis il est diffusé à tous ses membres. On y fait ensuite référence par son nom symbolique, identique sur tous les nœuds membres.

Le pilote `multi` est un pilote *Puk* pour gérer un type de module. Son but est de gérer le chargement sur plusieurs nœuds de modules avec une synchronisation de type SPMD sur un *groupe* de nœuds. Les unités d'un module de type `multi` sont eux-mêmes des modules, comme pour le pilote `pkg`. Un module de type `multi` doit obligatoirement comporter un attribut de module de nom `GroupID` contenant le nom d'un groupe existant. Le pilote `multi` réalise ensuite les diverses opérations *Puk* (`puk_load`, `puk_start`, etc.) de façon synchrone sur tous les nœuds lorsque l'utilisateur les appelle sur un seul nœud.

9.3.3 Connaissance de la topologie et sélection automatique : *NetSelector*

La sélection et l'assemblage des adaptateurs dans PadicoTM sont confiés à une entité appelée *NetSelector*, qui se décompose en trois parties : une partie centrale gère la sélection de méthode, et deux parties dédiées aux *VLink* et aux *Circuit* gèrent l'assemblage des modules qui présentent leur interface abstraite respective.

Padico *NetSelector* implémente les mécanismes de sélection automatique des méthodes de communication et adaptateurs. Il fournit des informations sur la topologie du réseau, stocke la *collection* d'adaptateurs ainsi que les préférences enregistrées par l'utilisateur.

9.3.3.1 Description des ressources

Chaque nœud a connaissance de la topologie de son environnement réseau. La vision de la topologie est propre à un nœud. La topologie est décrite à l'aide de trois types de structures : `host_t` décrit un hôte, c'est-à-dire une machine physique, `node_t` décrit un nœud, hébergé par un hôte. Plusieurs nœuds peuvent coexister sur un hôte. `network_t` décrit un réseau, entité qui interconnecte des hôtes ou des nœuds. Un qualificatif de *portée* précise son type parmi `scope` = *host* (cas de IP : une adresse qualifie une machine) ou `scope` = *node* (cas de Madeleine : une adresse qualifie un nœud). La figure 9.14 représente un exemple de topologie selon le modèle *nœud/hôte/réseau* introduit à la section 6.6. Un seul des nœuds a accès à *internet*, le réseau qui désigne toutes les machines adressables par IP qui ne font pas partie de la topologie connue. Cette topologie sera vue par le nœud *node0* sous la forme réprésentée à la figure 9.15 — les cadres symbolisent des structures, les flèches symbolisent des pointeurs. Les `bindings` sont des listes de liaisons entre un nœud ou un hôte et un réseau. Chaque liaison est un couple constituée d'un pointeur vers la structure décrivant le réseau et de l'adresse associée. Par exemple, vers un réseau IP un `binding` est qualifié par une adresse IP, vers un réseau Myrinet il est qualifié d'un numéro.

Les liens entre les méthodes logicielles d'accès aux réseaux et les descriptions des réseaux eux-mêmes sont réalisés par des structures de type `netaccess_t`. Chaque structure `netaccess_t` décrit

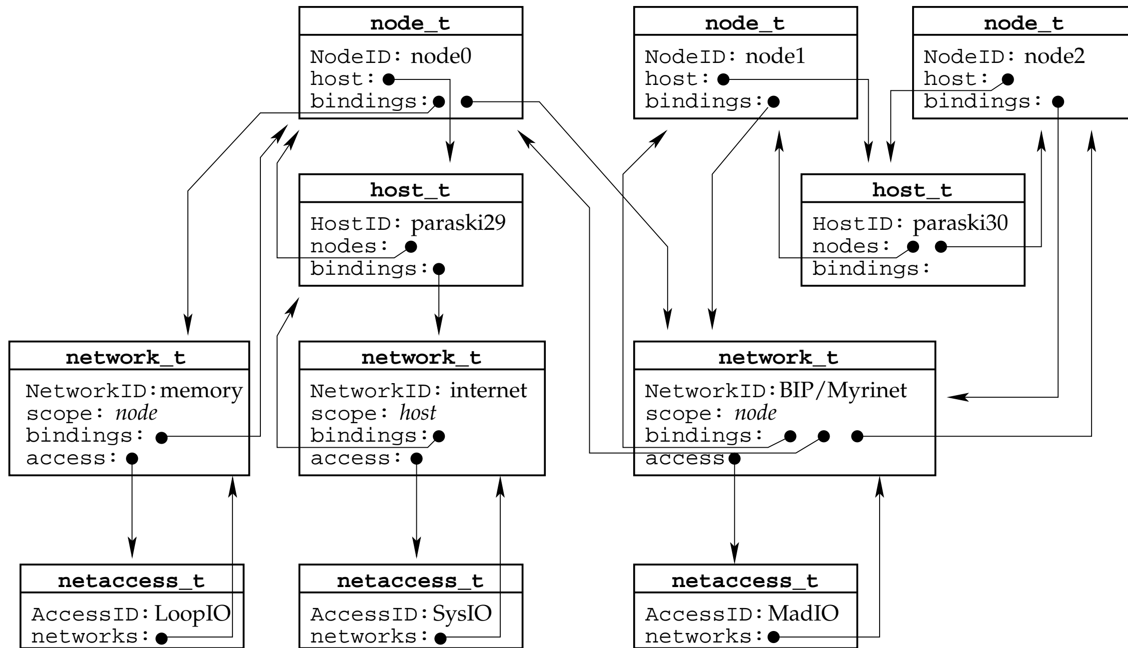

FIG. 9.15 – *Structures PadicoTM correspondant à la topologie de la figure 9.14 du point de vue du nœud* node0.

```
typedef struct {
  char*         AdapterID;
  iface_t       provides;
  iface_list_t  uses[];
  method_t      implements;
  void*         implementation;
} adapter_t;
```

```
typedef struct {
  char*  IfaceID;
  adapter_list_t providers;
} iface_t;
```

FIG. 9.16 – *Structures de description des adaptateurs.*

une méthode d'accès au réseau de *NetAccess* : *MadIO*, *SysIO*, ou *LoopIO* sur l'exemple de la figure 9.15, *LoopIO* fournissant des connexions en *loopback* à l'intérieur d'un processus. À l'initialisation des environnements réseaux lors du chargement de PadicoTM, les différents modules de *NetAccess* déclarent leur lieur `netaccess_t` et les réseaux `network_t` qu'ils gèrent. Ils énumèrent ensuite la topologie qu'ils connaissent pour construire les descripteurs d'hôtes et de nœuds stockés dans *NetSelector*.

Le *NetSelector* a également connaissance de la collection d'adaptateurs. Un adaptateur est représenté par une structure de type `adapter_t` telle que décrite à la figure 9.16. Un telle structure contient le type d'interface abstraite offerte et les interfaces utilisées, la méthode de communication implémentée, et un pointeur sur une description de l'implémentation réelle de l'adaptateur. Cette description de l'implémentation dépend du type d'interface offerte : dans le cas d'une interface *VLink*, ce champ contient un ensemble de fonctions du type présenté à la figure 9.8 ; dans le cas d'une interface *Circuit*, il s'agit d'un ensemble de fonction du type de la figure 9.12. La figure 9.17 représente l'exemple des adaptateurs de la collection principale — *ie.* dont la méthode de communication est "direct", sans transformation — qui offrent une interface *VLink*.

9.3.3.2 Canal de contrôle

Les opérations de sélection automatique nécessitent parfois un dialogue entre les nœuds qui se connectent. De plus, certains adaptateurs ont besoin d'une négociation de certains paramètre lors de la connexion — par exemple les flux parallèles demandent d'être d'accord sur le nombre de flux. Pour ces *communications de contrôle*, PadicoTM offre un canal de contrôle. Il est géré par le *NetSelector* et est destiné à être utilisé pour la sélection et mis à disposition des adaptateurs pour leurs communications

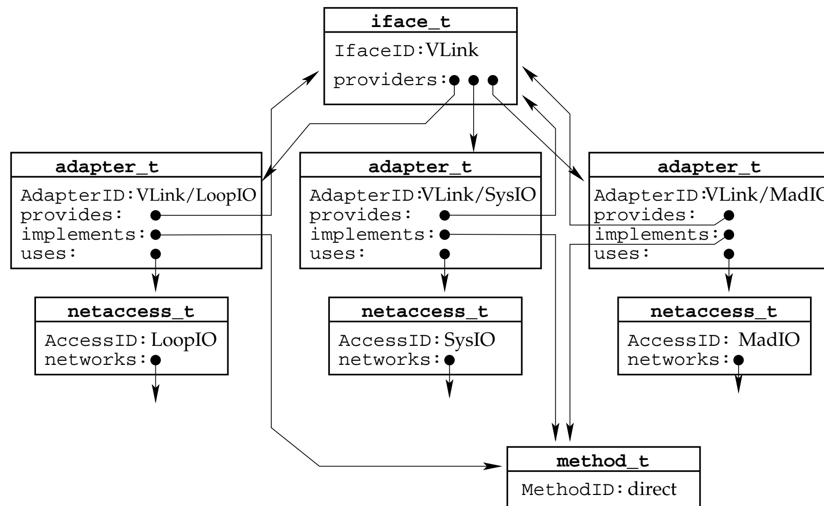

FIG. 9.17 – *Représentation de la partie de la collection principale d'adaptateurs offrant l'interface* VLink.

d'administration. Les nœuds accessibles par un canal de contrôle définissent le *monde connu*.

Les canaux de contrôle sont construits automatiquement par le *NetSelector*, dès la prise en charge d'un réseau. C'est le premier canal ouvert sur chaque réseau. Un canal de contrôle est automatiquement construit sur *MadIO* au démarrage des processus sur une grappe. La construction d'un canal de contrôle *SysIO* entre deux grappes *MadIO* (un *bridge*) doit être réalisée explicitement par l'utilisateur. Pour celà, il faut *présenter* les grappes l'une à l'autre en demandant une référence à l'une et en la donnant à l'autre. À l'établissement du *bridge*, les deux parties s'échangent leurs connaissances de topologie et les propagent aux nœuds qu'ils connaissent.

9.3.3.3 Sélection et assemblage

La sélection automatique et l'assemblage sont réalisés par des parties spécifiques pour chaque abstraction, *VLinkSelector* pour *VLink* et *CircuitSelector* pour *Circuit*. Ces deux sélecteurs se basent sur la topologie et les préférences gérées par le *NetSelector* global.

Le *VLinkSelector* gère l'assemblage et les demandes de sélection automatique d'adaptateurs pour fournir une interface *VLink*. Quand une application appelle les primitives `vlink_*`, c'est en réalité le *VLinkSelector* qui est appelé et qui redirige vers les primitives de l'adaptateur adéquat en fonction de l'état mémorisé dans le descripteur. Il gère en particulier les appels à `vlink_connect`, choisit l'adaptateur, et appelle ensuite la primitive correspondante de l'adaptateur. Une requête de sélection pour un `vlink_connect` a lieu entre deux *points de connexion* qui sont dans ce cas des `vlink_t`. En effet, la connexion se fait vers un descripteur déjà ouvert en écoute sur la machine distante. Le sélecteur transmet la requête au *VLinkSelector* distant *via* le canal de contrôle ; le sélecteur distant se charge de charger, initialiser, et assembler les adaptateurs nécessaires de son côté pour construire le lien. Lors des appels à la primitive `vlink_create`, le *VLinkSelector* crée un descripteur vide. Au moment du `vlink_bind`, il crée des descripteurs correspondant à une liste d'adaptateurs chargés par défaut. Par exemple *VLink/SysIO* est un adaptateur chargé par défaut, ce qui assure qu'à chaque *VLink* créé comme serveur correspond effectivement une *socket* physique en écoute et assure l'interopérabilité. *VLink/MadIO* est également chargé par défaut pour accélérer les connexions et court-circuiter la requête sur le canal de contrôle quand on utilise directement un réseau rapide. L'assemblage proprement dit se résume alors à mémoriser dans le `vlink_t` quel est l'adaptateur qui le gère.

La figure 9.18 illustre ce mécanisme pour *VLink/MadIO*. Initialement, *VLink/SysIO* et *VLink/MadIO* sont chargés par défaut de part et d'autre. Le client (à droite) demande une connexion. Le *VLinkSelector* s'adresse au *NetSelector* pour obtenir une connexion vers l'adresse demandée (adresse IP et numéro de port — port=1830 dans l'exemple). Le *NetSelector* applique les règles, et choisit *MadIO* ; il envoie

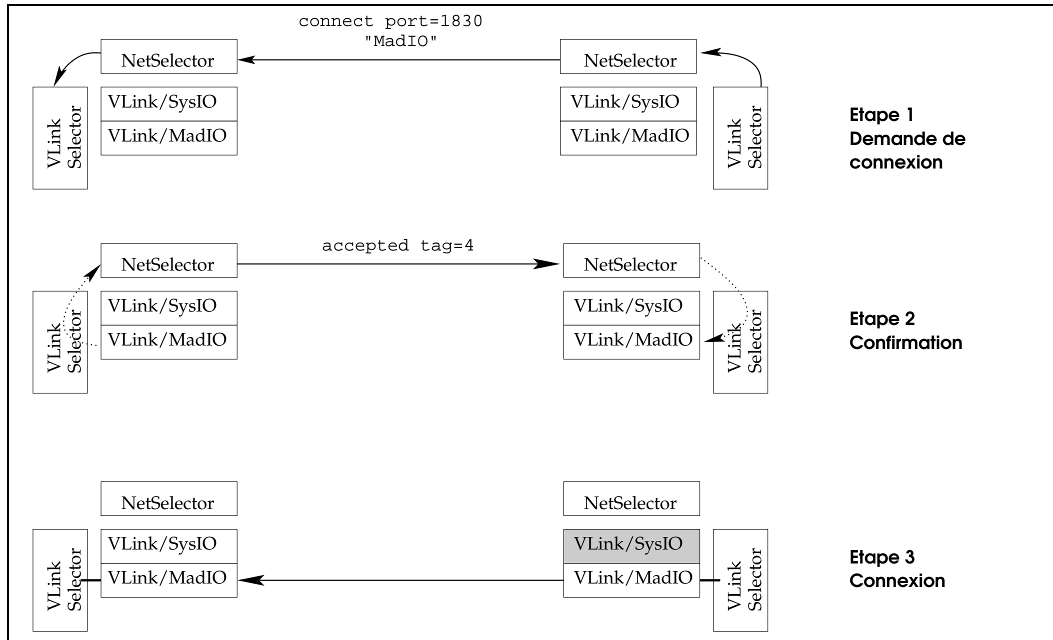

FIG. 9.18 – *Exemple de déroulement de sélection automatique : sélection de* MadIO*.*

une requête à son homologue sur le pair distant *via* le canal de contrôle ; la requête est transmise au *VLinkSelector* qui est en charge du *VLink* enregistré à l'adresse port=1830 (étape 1). Le *VLinkSelector* vérifie que *VLink/MadIO* est chargé, obtient son adresse par le champ `bind` de l'observateur, et fait suivre l'adresse retournée à l'adadaptateur du client *via* le canal de contrôle du *NetSelector* (étape 2). Enfin, le *VLink/MadIO* du client établit la connexion avec le *VLink/MadIO* du serveur ; ce dernier crée alors un nouveau *VLink* qu'il place dans la file des connexions en attente et envoie un signal `listener` à son observateur. Le client envoie un signal `remote` à son observateur. Les deux *VLinkSelector* mémorisent le choix, et celui côté client désactive les adaptateurs devenus superflus.

En présence de plusieurs nœuds sur le même hôte, le *NetSelector* envoie la demande de connexion (étape 1) à tous les nœuds de l'hôte. Un seul au maximum a enregistré un *VLink* sur le port demandé, donc un seul répond positivement à la demande de connexion. Si aucun ne répond positivement, ceci indique que le nœud serveur n'est pas dans un processus géré par PadicoTM ; dans ce cas, le *NetSelector* choisit le réseau *internet* par *VLink/SysIO*. Ces négociations sont cachées dans le *NetSelector* ; le *VLinkSelector* reçoit l'adresse de connexion après les négociations.

Notons que dans le cas où la demande de connexion est formulée directement en système d'adressage `AF_MADIO`, le *VLinkSelector* n'effectue aucune résolution d'adresse et la passe directement à *VLink/MadIO*. Dans ce cas, les étapes 1 et 2 n'existent pas ; la connexion démarre directement à l'étape 3.

La figure 9.19 présente le mécanisme de connexion interopérable avec un hôte *internet* standard. L'adresse du *VLink* (port=1830 dans l'exemple) correspond réellement à l'adresse de la *socket SysIO* utilisée par *VLink/SysIO*. Un hôte *internet* standard établit dont directement sa connexion à cette adresse (étape 1). L'adaptateur *VLink/SysIO* détecte la connexion, la place dans la file de connexions en attente, et envoie un signal `listener` à son observateur (étape 2).

La figure 9.20 illustre le déroulement d'une connexion *ParallelStreams*. À la première étape, le *VLinkSelector* client envoie une requête à son homologue dans le serveur *via* le canal de contrôle du *NetSelector*. Le choix du *NetSelector* s'est porté sur l'adaptateur *ParallelStreams*. Le *VLinkSelector* du serveur charge alors *ParallelStreams* et crée un *VLink* l'utilisant ; il renvoie les informations retournées par le `vlink_bind` de cet adaptateur au client (étape 2). Ceci permet au client de désactiver les adaptateurs devenus inutiles pour ce *VLink* et de lancer la connexion avec les paramètres reçus (étape 3).

Le *CircuitSelector* s'occupe de l'assemblage et de la sélection pour les adaptateurs qui fournissent une interface *Circuit*. Il ne gère que la construction ; quand un circuit est en place, les personnalités

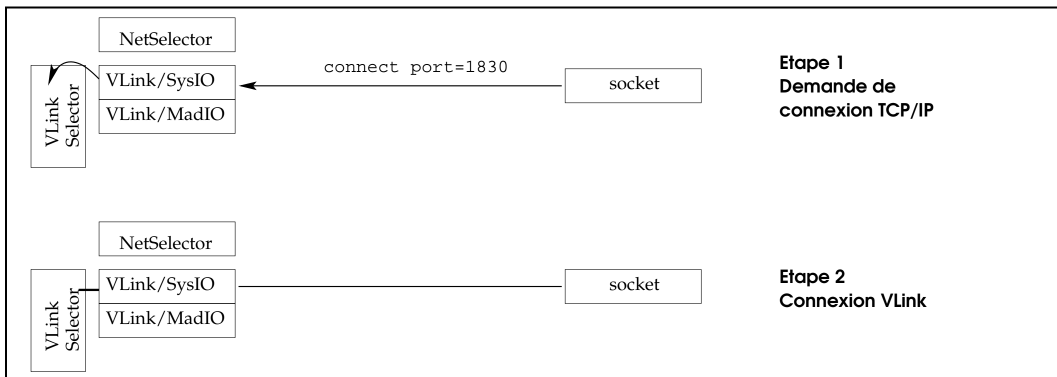

FIG. 9.19 – *Exemple de déroulement de sélection automatique : interopérabilité avec* TCP/IP.

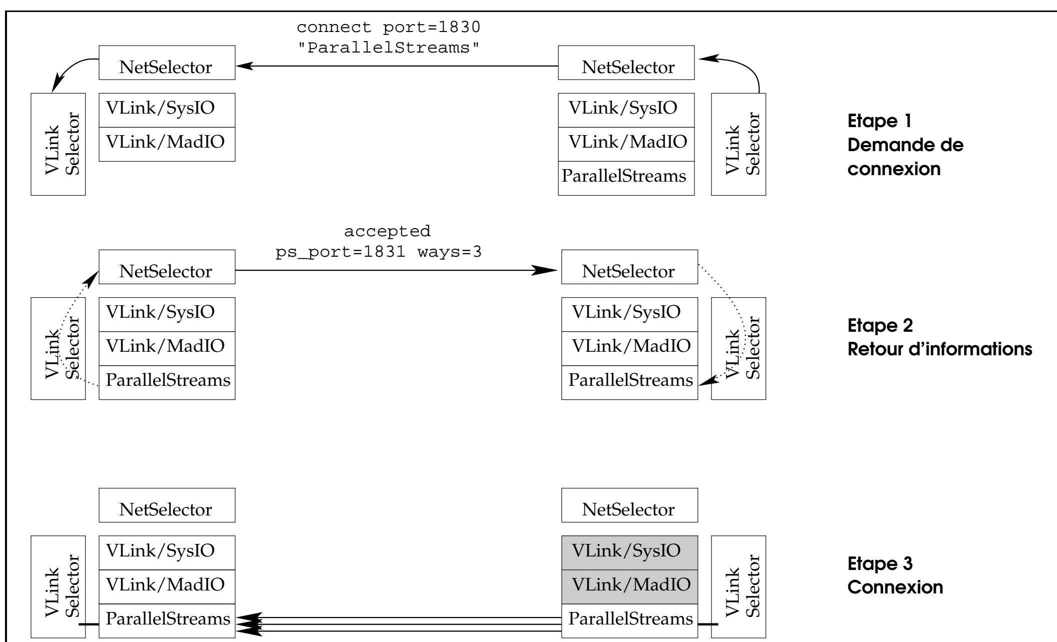

FIG. 9.20 – *Exemple de déroulement de sélection automatique : utilisation d'un adaptateur alternatif.*

appellent directement les fonctions de l'adaptateur. Pour la création d'un circuit, il reçoit un ensemble de nœuds sous la forme d'un `group_t`. Le *CircuitSelector* calcule alors le réseau à emprunter pour chacune des routes à l'intérieur du circuit en fonction des préférences ensuite, il regroupe les liens gérés par un même adaptateur s'il a déclaré être capable de gérer des groupes de plus de deux nœuds. Le résultat est un ensemble de groupes de nœuds, chaque groupe étant associé à un adaptateur. S'il y a plusieurs groupes, l'adaptateur *Circuit* composite est placé au-dessus pour les regrouper. L'algorithme est exécuté sur le premier nœud du circuit ; il propage ensuite ses résultats par le canal de contrôle aux autres nœuds du circuit.

Les *VLinkSelector* et *CircuitSelector* peuvent interagir dans le cas où une requête met en jeu plusieurs interfaces. Par exemple, si un circuit emprunte un lien qui utilise la compression fournie sous forme d'adaptateur *VLink*, le *CircuitSelector* sélectionne l'adaptateur croisé abstrait *Circuit/VLink* et confie l'assemblage de ce qui est sous cet adaptateur au *VLinkSelector*.

Les deux sélecteurs spécialisés se basent sur les mécanismes de décision et les préférence du *NetSelector* global. Au final, chaque décision de sélection revient à trouver un chemin dans les structures présentées aux figures 9.15 et 9.17, entre deux ou plus `iface_t` sur des nœuds différents. Les préférences sont données sous forme de reconnaissance de motif (*pattern matching*) sur les chemin et indiquent quel réseau et/ou quelle méthode utiliser. Le sélecteur cherche ensuite un chemin qui emprunte le réseau voulu et les méthode choisies.

9.4 Adaptateurs d'abstraction

Nous présentons dans cette section quelques adaptateurs d'abstraction de PadicoTM. Il s'agit des adaptateurs les plus intéressants, à savoir *VLink/MadIO*, et de quelques adaptateurs généralisés : la compression adaptative, les flux parallèles, un tunnel, et un protocole à tolérance de pertes.

9.4.1 VLink/MadIO

VLink/MadIO est l'adaptateur trans-paradigme qui offre une interface abstraite de type réparti au-dessus des réseaux haute performance gérés par Madeleine. Les principales adaptations réalisées par *VLink/MadIO* concernent l'adressage, la dynamicité des connexions, et l'adaptation des primitives d'échange de message.

L'adaptateur *VLink/MadIO* propose sa propre classe d'adressage appelée `AF_MADIO` stockée dans une structure standard `struct sockaddr`. Une telle adresse contient l'identification du nœud en format *MadIO* — c'est-à-dire un simple numéro d'ordre —, et un identifiant de connexion (équivalent d'un numéro de port) appelé un *tag*. Bien entendu, *VLink/MadIO* accepte également les adresses en format standard *internet* `AF_INET` (adresse IP + port) et les mécanismes de traduction d'un système à l'autre. La façon courante d'utiliser cet adaptateur est de manipuler des adresses `AF_INET` et de laisser l'adaptateur faire les conversions. La conversion d'une adresse du système *internet* vers le système *MadIO* est habituellement triviale en présence des structures de description de topologie ; cependant, quand plusieurs nœuds cohabitent sur un hôte (et ont donc la même adresse IP), il est nécessaire d'envoyer une requête de contrôle à chacun de ces nœuds pour savoir lequel répond au numéro de port demandé. La portée des adresses *internet* est en effet un hôte (`scope` = *host*) dans les structures `network_t`) alors que la portée des adresses *MadIO* est un nœud (`scope` = *node*).

VLink/MadIO apporte également la dynamicité des connexions au-dessus des canaux *MadIO* à topologie statique. Un *VLink* est identifié par le récepteur grâce à un *tag*. Établir une connexion revient donc à allouer un *tag* sur les deux nœuds ; les deux *tag* peuvent être différents de part et d'autre, il suffit de savoir quel *tag* est associé à la connexion pour le pair récepteur. *VLink/MadIO* alloue un *tag* lors d'un appel à `vlink_bind`. Le descripteur `vlink_t` est alors enregistré dans une table d'équivalence entre *tag*, `vlink_t` et numéro de port (si l'adresse a été fournie en format `AF_INET`). Lors d'un `vlink_connect`, *VLink/MadIO* envoie une requête avec le *tag* du client et l'adresse demandée (*tag* ou numéro de port) ; le serveur crée un nouveau *VLink* et répond avec un accusé de réception qui contient le *tag* du serveur. La fermeture a lieu de la même manière, en deux temps, avec une requête provenant d'un pair et un accusé de réception quand la fermeture est effective.

L'échange de messages par *MadIO* est basé sur des messages actifs, c'est-à-dire à traiter dès qu'ils arrivent. Un message peut donc arriver sur un lien alors qu'aucun `vlink_recv` n'a été invoqué. *VLink/MadIO* met donc en place une boîte au lettre de réception pour stocker ces messages reçus alors que le destinataire n'est pas encore prêt. Cependant, l'usage de ces boîte aux lettres pénalise les débits car il induit une copie des données. Au-delà d'un certain seul, nous utilisons donc un mécanisme classique de *rendez-vous* : quand un émetteur veut envoyer un grand message, il envoie d'abord une demande d'autorisation d'émission. Le récepteur envoie l'autorisation uniquement quand le `vlink_recv` a été invoqué et qu'il est prêt à recevoir. De cette façon, toute réception au-delà d'une certaine taille est faite en zéro copie. De plus, une accumulation de petits messages dans une boîte aux lettres de réception peuvent saturer la mémoire du récepteur ; pour l'éviter, *VLink/MadIO* implémente un contrôle de flux rudimentaire : quand la boîte de réception franchit une certaine taille totale, le récepteur envoie un message qui demande à l'émetteur de ne plus émettre. Il envoit un message qui le réautorise quand la boîte de réception diminue en taille. *VLink/MadIO* adapte également la gestion des frontières de messages. La déclinaison qui fournit un protocole de type `SOCK_STREAM` (comme TCP/IP) est basée sur des flux continus. La gestion des flux continus a une influence essentiellement sur les rendez-vous : une autorisation d'émission après un rendez-vous peut ne demander qu'une partie des données proposées. La plupart du temps, il est possible de gérer les flux en zéro copie.

VLink/MadIO présente un certain nombre de paramètres à régler en fonction de l'utilisation. Le paramètre principal est le seuil pour choisir entre un envoi direct et un envoi par rendez-vous. Un rendez-vous ajoute un aller-retour, soit deux latences réseau ; un envoi sans rendez-vous risque une copie en mémoire. Sur une machine récente équipée de Myrinet-2000, deux latences représentent environ 12 μs, ce qui correspond au temps pour copier environ 16 Ko. Sur d'autres machines équipées d'un réseau de la même génération que la machine, on retrouve le même ordre de grandeur de seuil. Nous positionnons donc par défaut le seuil à cette valeur.

9.4.2 Exemples d'adaptateurs généralisés

Dans cette section, nous présentons quelques exemples d'adaptateurs généralisés. Ces adaptateurs apportent des méthodes de communication supplémentaires. Ils complètent les adaptateurs de la collection principale.

Adaptateur *VLink*/tunnel. Un *tunnel* fait passer une connexion de bout-en-bout sur une chaîne constituée de plusieurs tronçons, par exemple des liens inter- et intra-grappe. Il est basé sur un *bridge*, lien de contrôle construit explicitement par l'utilisateur. Lors de sa construction, un *bridge* déclare au *NetSelector* un nouveau réseau, de type `network_t`, représentant un réseau virtuel constitué de l'interconnexion des mondes connus des deux nœuds reliés par le *bridge*. Il déclare également la méthode d'accès `tunnel` de type `netaccess_t` à ce réseau. La méthode usuelle de l'utiliser est de passer par l'adaptateur *VLink*/tunnel.

Il va de soi qu'un tel mécanisme ne doit être utilisé que quand la connectivité directe n'est pas possible et qu'il n'existe pas d'autre moyen plus performant. Un tunnel est utilisé habituellement pour un canal de contrôle entre plusieurs grappes ; il est par exemple utilisé pour contrôler la construction d'un *Circuit* qui recouvre plusieurs grappes. Il n'est utilisé pour transporter des données qu'en dernier ressort, avec une priorité plus faible que le réseau par défaut *internet*.

Compression adaptative : AdOC. La compression dans PadicoTM est présente sous la forme d'un adaptateur *intercepteur* réparti, c'est-à-dire un adaptateur qui offre une interface abstraite de type réparti et utilise une interface du même type. Nous avons utilisé la bibliothèque AdOC [114], basée sur la `zlib`. AdOC a la particularité d'adapter le taux de compression en fonction des vitesses respectives du processeur et du lien réseau. L'adaptation d'une interface *VLink* est relativement aisée. En effet, AdOC présente une interface inspirée de l'interface des *sockets* et est intrinsèquement *multi-thread*, ce qui facilite l'adaptation à une interface asynchrone.

Flux parallèles sur WAN : *ParallelStreams*. Une façon d'accélérer les transmissions sur les WAN à latence élevée consiste en l'utilisation de plusieurs flux parallèles pour transporter les données d'un flux logique. Une telle approche est dépendante du réseau sous-jacent ; comme proposé en section 6.5.3, l'adaptateur *ParallelStreams* est une variante de l'adaptateur *VLink/SysIO*. Il a été mis en œuvre par Joel Daniels lors de son stage [110] de *Bachelor of Sciences* effectué au sein du projet PARIS. Les *ParallelStreams* sont donc implémentés sous la forme d'un adaptateur qui utilise *SysIO* et offre une interface *VLink*. Cet adaptateur présente deux paramètres : le nombre de flux et la taille des blocs (facteur d'entrelacement entre le flux).

Protocole à tolérance de perte réglable : VRP. Sur les réseaux qui présentent un taux de pertes élevé, la fiabilisation mise en œuvre par TCP coûte cher. Certaines applications préfèrent concéder une partie de fiabilité pour obtenir en échange de meilleures performances. Il s'agit alors de *tolérance de pertes*. Cependant, il est rare que le niveau de garantie offert par UDP— pertes imprévisibles et incontrôlables — soit satisfaisant. Il existe souvent des données de contrôle, des en-têtes ou des parties critiques que l'on ne veut pas perdre. De plus, il peut être souhaitable de contrôler la répartition des pertes dans les messages.

Nous introduisons le protocole VRP [14] (*Variable Reliability Protocol*), un protocole avec une tolérance de perte réglable, implémenté au-dessus de UDP et qui fournit des contrôles et garanties supplémentaires par rapport à un accès UDP direct. Ces garanties sont exprimées par deux paramètres :

- le taux de pertes tolérables maximum, exprimé sous la forme d'un pourcentage ;
- la longueur de perte consécutive maximum, qui contrôle la granularité des pertes.

Il est possible de spécififer des paramètres différents pour plusieurs fragments par message. L'application spécifie ses paramètres de tolérance de pertes, VRP se charge de les faire respecter et ré-émet certains paquets si nécessaire. Le récepteur, quand il reçoit un message, est informé des pertes effectives dans le message.

VRP est implémenté sous la forme d'un adaptateur basé sur *SysIO*. En revanche, ses propriétés ne sont pas exprimables par l'interface *VLink* ni *Circuit*. En effet, l'API de VRP met eu jeu des propriétés qui lui sont propres pour contrôler la tolérance de pertes. VRP offre donc une interface spécifique, et doit être programmé explicitement par l'application. Le *NetSelector* ne prend *jamais* l'initiative d'utiliser VRP si on ne lui demande pas explicitement.

Les algorithmes mis en œuvre dans VRP sont décrits dans [19, 14]. Le principal obstacle auquel se heurte actuellement VRP est le filtrage complet des paquets UDP par la plupart des pare-feux. VRP étant basé sur UDP, les sites protégés par de tels pare-feux ne peuvent être accédés par des liens VRP, ni en connexion entrante, ni en connexion sortante.

9.5 Personnalités et exécutifs sur PadicoTM

Nous présentons dans cette section les personnalités que PadicoTM fournit au-dessus des interfaces abstraites, et les exécutifs qui utilisent ces personnalités. La liste donnée peut être facilement étendue pour ajouter d'autres personnalités et exécutifs.

9.5.1 Personnalités du réparti

PadicoTM comprend trois personnalités du réparti au-dessus de l'interface *VLink* :

VIO — VIO (pour *Virtual Input/Output*) est une personnalité qui présente une interface très similaire à l'interface *sockets* BSD et fichiers standard au-dessus de *VLink* et *VFile*. Chaque primitive de l'interface standard *socket* a son équivalent en `vio_*` (*eg.* `vio_connect` est la fonction VIO équivalente à `connect`). Les modes d'accès bloquants et non-bloquants sont implémentés. VIO implémente des observateurs secondaires capables de s'enregistrer dans les observateurs primaires, pour fournir des primitives `select()` et `poll()` efficaces. VIO offre quelques extensions par rapport aux *sockets* BSD ; par exemple, `vio_cancel` permet d'annuler une opération bloquante en cours avant sa terminaison normale.

Intercepteurs de fonctions *SysW* — VIO offre une interface très semblable aux interfaces standard, mais il demande à être utilisé explicitement. La personnalité *SysW* (*system wrappers*) permet d'intercepter les appels correspondants lors de l'édition de liens, et de les détourner automatiquement vers leur équivalent VIO. Un code peut alors utiliser VIO sans modification du source ni recompilation, simplement au moment de l'édition de liens.

Posix AIO — L'interface Posix AIO est le standard d'interface asynchrone pour les accès aux fichiers et aux réseaux. Cette interface ne concerne que les opérations de lecture/écriture ; la gestion des liens et la construction des descripteurs est réalisée par les interfaces habituelles. L'interface AIO gère l'asynchronisme par des primitives qui démarrent une opération ; l'état d'avancement est consulté par scrutation en appelant la fonction `aio_return`, ou peut déclencher un signal `SIGIO` si demandé. L'interface Posix AIO étant similaire mais plus rudimentaire que l'interface *VLink*, il est aisé de réaliser une personnalité AIO/*VLink*.

9.5.2 Personnalités du parallélisme à mémoire distribuée

PadicoTM comprend deux persoannlités pour le parallélisme à mémoire distribuée implémentées au-dessus de *Circuit*. Ces personnalités sont :

FastMessage 2.0 — *Fast Messages* 2.0, décrit en section 2.2.3, propose une interface basée sur cinq fonctions. L'envoi est basé sur des messages à construction incrémentale, et la réception se fait par message actifs. C'est une interface très proche de *Circuit*. La personnalité *Fast Messages* 2.0/*Circuit* comprend 50 lignes de code en tout.

Madeleine — PadicoTM propose une interface Madeleine *au-dessus* de *Circuit*. Il s'agit en réalité d'un *script* qui transforme un code Madeleine en code pour *Circuit*. Les informations de topologie étant habituellement lues directement dans les structures internes de Madeleine par les application, les parties correspondantes ne peuvent pas être traitée automatiquement et doivent donc être adaptées "à la main" en conséquence.

9.5.3 Exécutifs sur PadicoTM

Dans cette section, nous décrivons les exécutifs qui ont été portés sur les personnalités de PadicoTM. Pour la plupart, l'intégration est directe grâce aux personnalités qui leur présentent l'interface qu'ils attendent.

MPI. Nous avons basé notre version PadicoTM de MPI sur l'implémentation MPICH/*Madeleine* [46] développée par Guillaume Mercier. MPICH/*Madeleine* est une variante prévue pour utiliser Madeleine 3 de l'implémentation MPI extensible MPICH [102]. Nous avons modifié MPICH/*Madeleine* pour qu'il utilise *Circuit* au lieu d'utiliser directement Madeleine. Le changement est opéré en grande partie de façon automatique par la personnalité Madeleine de *Circuit*. Le changement le plus important concerne la gestion de la topologie. À l'origine, MPICH/*Madeleine* extrait les informations de topologie (taille, canaux, etc.) de structures internes de Madeleine ; nous avons modifié cette partie pour que les informations de topologie soient simplement obtenues en interrogeant l'API adéquate de *Circuit*. Nous avons de plus réécrit une commande `mpirun` qui utilise les primitives *Puk* et son pilote `multi` pour l'exécution SPMD. Au total, les modifications concernent moins de 100 lignes du code original de MPICH/*Madeleine*.

CORBA. Nous avons porté diverses implémentations CORBA sur PadicoTM. Toutes sont basées sur des *sockets* BSD. Elles peuvent donc être utilisées telles quelles grâces à la personnalité *wrappers systèmes* au-dessus de *VLink*. Les implémentations testées sont : OmniORB 3 [45] des laboratoires AT&T de Cambridge, OmniORB 4 d'*Apasphere Ltd.*, MICO [25], et ORBacus TM [27] de *Iona*. Les deux versions d'OmniORB et les versions récentes de MICO font appel au *multi-threading*. L'interception des mécanismes de *multi-threading* vers Marcel est assurée par des scripts qui font la conversion automatiquement. De ce fait, aucune modification ne doit être faite manuellement dans ces implémentations CORBA pour les rendre utilisables sur PadicoTM.

Java Nous avons modifié la machine virtuelle Java "Kaffe" pour qu'elle utilise les services de PadicoTM. Des modifications ont été nécessaires car Kaffe fait une utilisation très particulière des signaux Unix et du *multi-threading*, utilisation qui est fondamentalement incompatible avec Marcel. Le code correspondant a dû être adapté. Ce travail a été réalisé par Benoît Hubert. Avec la version modifiée, les communications issues de Java (*sockets* et RMI) empruntent *VLink*. Il est ainsi possible de charger dans le même processus des codes Java en même temps que des codes écrits en C ou en FORTRAN. Il serait intéressant de refaire les mêmes manipulations avec la machine virtuelle Java de *Sun Microsystems*, implémentation de référence complète. Cette manipulation est cependant interdite par les termes de la license de Java ; nous n'avons donc pas pu tester cette machine virtuelle Java sur PadicoTM.

Mémoire virtuellement partagée : MOME. La mémoire virtuellement partagée MOME d'Yvon Jégou (projet PARIS, IRISA) a été portée sur PadicoTM en utilisant la personnalité *SysW* de *VLink*. Étant donnée l'existence d'une version de MOME sur Madeleine, il est envisageable de réaliser une version de MOME sur *Circuit*.

Autres exécutifs de type réparti. Outre plusieurs implémentations CORBA, nous avons porté d'autres exécutifs de type réparti sur PadicoTM :

- gSOAP est une version de SOAP qui propose une projection vers le langage C. Les communications sont assurées par le protocole HTTP qui exploite des *sockets* BSD. L'adaptation est immédiate, sans modifier la moindre ligne de code, grâce aux *wrappers systèmes*.
- *Certi* [147] est une implémentation du standard HLA [109]. *Certi* utilise des *sockets* TCP/IP et des *sockets* Unix, deux types immédiatement prix en charge par les *wrappers systèmes*.
- ICE TM est un intergiciel de la société *ZeroC*. Nous l'avons présenté au chapitre 2. ICE est annoncé comme une évolution de CORBA et en est donc très proche. Les même techniques que celles utilisées pour les implémentations CORBA permettent d'utiliser ICE sur PadicoTM.

9.6 Conclusion

Nous avons présenté la plate-forme PadicoTM qui est une implémentation de notre modèle de plate-forme de communication multi-paradigme pour les grilles de calcul. PadicoTM est constitué d'environ 37 000 lignes de code C (soit environ 1 Mo de sources). PadicoTM contient un cœur capable d'arbitrer les accès aux réseaux accessibles par Madeleine et par *sockets*. Au-dessus, il fournit les abstractions *VLink* et *Circuit* respectivement pour les communications de type réparti et de type parallèle à mémoire distribuée, chacune implémentée sur les deux types de réseaux et sur la mémoire partagée (*loopback*). En complément, diverses méthodes de communication spécialisées sont fournies sous forme d'adaptateurs supplémentaires, comme *ParallelStreams*, AdOC, VRP. Les personnalités présentent une interface adaptable qui permet d'utiliser des exécutifs existants. PadicoTM supporte des exécutifs variés, à savoir une implémentation MPI, plusieurs implémentations CORBA, une implémentation SOAP, la MVP MOME, une implémentation HLA, et les exécutifs ICE et Kaffe.

Le logiciel PadicoTM a été rendu public sous licence GNU et déposé à l'Agence pour la Protection des Programmes (APP) sous le numéro IDDN.FR.001.260013.000.S.P.2002.000.10000. Il a également été retenu comme contribution au consortium *Gelato* [112]. PadicoTM est disponible au téléchargement à l'adresse `http://www.irisa.fr/paris/Padicotm/`.

Chapitre 10

Évaluation de PadicoTM

Sommaire

10.1 Micro-benchmark de *NetAccess* 145
10.1.1 *NetAccess MadIO* 145
10.1.2 *NetAccess SysIO* 147
10.1.3 Arbitrage 148
10.2 Évaluation des abstractions *VLink* et *Circuit* 150
10.3 Performances des exécutifs sur PadicoTM 150
10.3.1 MPI 150
10.3.2 CORBA 151
10.3.3 Autres exécutifs 153
10.3.4 Comparaison et discussion 154
10.4 Méthodes de communication pour WAN 155
10.4.1 ParallelStreams 155
10.4.2 AdOC 156
10.4.3 VRP 157
10.5 Conclusion 157

Dans ce chapitre, nous présentons une évaluation des performances de PadicoTM : nous mesurons les performances de *NetAccess* et nous les comparons aux méthodes d'accès sans arbitrage, nous présentons les performances des adaptateurs et personnalités, et nous étudions les performances obtenues par les divers exécutifs dans PadicoTM.

10.1 Micro-benchmark de *NetAccess*

Dans cette section, nous étudions les performances de *NetAccess MadIO* et *SysIO*. Notre plate-forme de test est constituée de bi-*Pentium III* 1 GHz équipés de Myrinet-2000 et Ethernet-100. Le test de base consiste en un *ping-pong* de message de longueur variable. Les résultats présentés sont des moyennes obtenues sur un grand nombre de test.

10.1.1 *NetAccess MadIO*

Débit et latence. Le débit obtenu par *NetAccess MadIO* sur Myrinet-2000 est représenté par le graphe de droite de la figure 10.1, en fonction de la taille des messages. L'évolution du débit est régulière jusqu'à un plafond à 241 Mo/s, soit plus de 96 % du maximum de 250 Mo/s permis par le matériel Myrinet-2000.

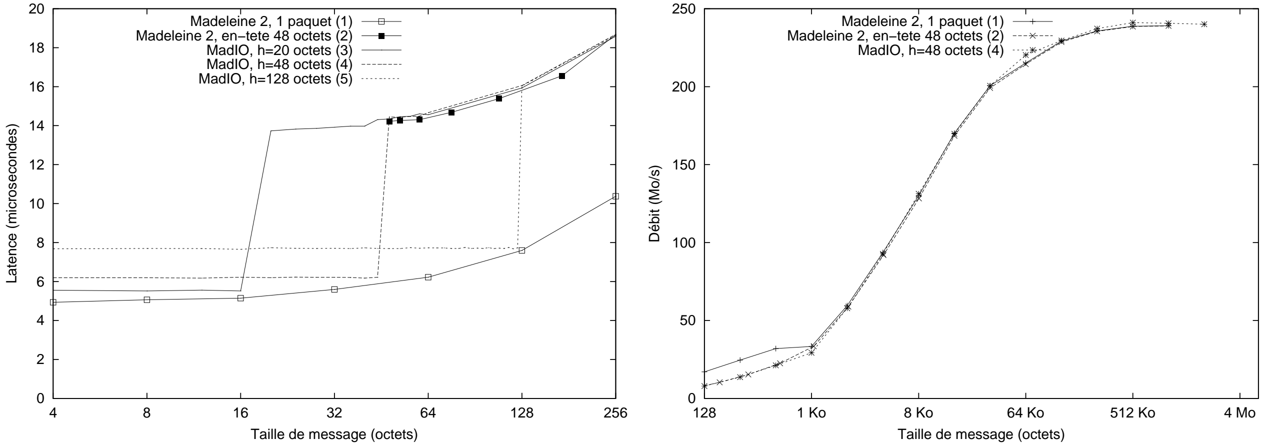

FIG. 10.1 – *Comparaison de* MadIO *et Madeleine sur Myrinet-2000 (à gauche : latence ; à droite : débit).*

Le temps de transfert obtenu par *NetAccess MadIO* sur Myrinet-2000 pour de petits messages est donné par le graphe de gauche de la figure 10.1, en fonction de la taille des données l. Nous donnons les courbes correspondants à plusieurs valeur du paramètre h, taille totale de l'en-tête agrégé *MadIO*. La taille d'en-tête indiquée comprend la clef de multiplexage de 4 octets ajoutée par *MadIO* ; la taille de message inclut l'espace restant disponible dans l'en-tête. Toutes les courbes du graphe de gauche de la figure 10.1 présentent le même profil : une partie à temps constant correspondant aux tailles de données qui peuvent être contenues entièrement dans le seul en-tête ($l \leq h - 4$), puis une partie correspondant aux tailles de données qui nécessitent l'envoi d'un deuxième paquet, c'est-à-dire quand le corps de message n'est pas vide ($l > h - 4$). Le paramètre h n'a une influence que sur la première partie de la courbe, là où elle est constante. À partir du moment où les données sont transmises en deux paquets, la taille totale transmise sur le réseau ne dépend pas de la taille d'en-tête ; logiquement, le temps de transfert obtenu sur cette partie de la courbe est le même quelle que soit la taille d'en-tête. Les latences obtenues pour la première partie de la courbe s'étagent, selon la valeur de h, de 5.5 μs pour 20 octets à 7.7 μs pour 128 octets.

Nous pouvons modéliser le temps brut de transmission sur le réseau sous la forme $T = \beta + l\tau$. Pour le réseau Myrinet-2000, nous prenons $\beta = 4.9\,\mu$s et $\tau = 21\,$ns/octet ; l'approximation ainsi obtenue diverge de moins de 1 % par rapport aux mesures pour des tailles jusque 512 octets. Nous modélisons ainsi le temps T de transfert par *MadIO* en fonction de h et l par :

$$T(l,h) = \begin{cases} \beta_h & \text{si } l \leq h', \\ \beta_h + \beta_2 + (l - h')\tau & \text{si } l > h', \end{cases} \tag{10.1}$$

avec : $h' = h - 4$, taille disponible pour les données dans l'en-tête,
$\beta_h = \beta + h\tau$, temps de transmission de l'en-tête,
β_2, latence pour émettre un deuxième paquet — couramment, $\beta_2 = \beta$;
sur Myrinet-2000, $\beta_2 = 8.1\,\mu$s.

Le choix de h est un compromis entre la latence et la taille de données transmises en temps constant. Plus l'en-tête est grand, plus le saut provoqué par l'envoi d'un deuxième paquet est repoussé à une plus grande taille de données, mais plus la latence est élevée. Par exemple, pour h=20 octets, jusque 16 octets peuvent être transmis en un temps β_h=5.5μs, mais il faut alors 13.6 μs pour transmettre 20 octets. Pour h=128 octets, jusque 124 octets peuvent être transmis en un temps constant, mais la latence s'élève alors à β_h=7.7 μs même pour seulement 4 octets. Le critère principal de choix est donc la répartition des tailles de messages pour les petits messages et la taille d'en-tête des exécutifs. La formule correspondant à $l > h'$ se simplifie en $T(l,h) = \beta + \beta_2 + 4\tau + l\tau$ ce qui confirme qu'au-delà de $l > h'$ les performances ne dépendent pas de la valeur de h.

Comparaison avec Madeleine 2. La figure 10.1 présente une comparaison des latences (à gauche) et débits (à droite) de *MadIO* d'une part, et de Madeleine 2 en dehors de PadicoTM, mais en version *multi-thread* (c'est-à-dire avec Marcel), d'autre part. La version sans *threads* de Madeleine, qui obtient une latence plus faible, n'est pas présentée sur cette figure. La courbe (1) de Madeleine 2 correspond à un envoi constitué d'un seul `pack` ; ses performances sont excellentes. Toutefois, aucun exécutif à notre connaissance ne réalise ses envois en un seul paquet en toute circonstance. La plupart du temps, chaque message contient un en-tête de taille fixe, éventuellement suivi d'une partie variable. La courbe (2) correspond à un schéma plus réaliste d'utilisation de Madeleine par les exécutifs, avec un en-tête fixe (48 octets dans cet exemple) suivi d'un corps de message de longueur variable ; nous considérons que ces 48 octets font partie de la charge utile, d'où le démarrage de la courbe à $l = 48$ octets.

Les courbes de gauche représentent la latence. Les courbes (3) à (5) correspondent à *MadIO* avec h variant de 20 à 128 octets. Nous observons que le pallier à β_h pour les valeurs $l \leq h'$ est très proche du temps mesuré avec Madeleine 2 (courbe 1) pour la taille correspondante. Pour la partie après le saut, les courbes (3) à (5) restent très proches de la latence de Madeleine 2 avec en-tête (courbe 2). L'écart entre le temps de *MadIO* et le temps mesuré avec Madeleine 2 en reproduisant le schéma de communication est toujours inférieur à 370 ns. Ce surcoût correspond aux opérations de multiplexage/démultiplexage : multiplexage (allocation de l'en-tête, allocation d'un canal physique au canal logique), surcoût de la clef de multiplexage dans l'en-tête (adjonction de 4 octets à chaque message, soit $4 \times \tau = 85$ ns), démultiplexage (résolution de la clef en canal logique, et appel indirect de fonction). De plus, le démultiplexage étant réalisé par table de hachage, nous pouvons considérer que, pour un nombre raisonnable de canaux ouverts, il est effectué en temps quasi-constant.

Les courbes de droite de la figure 10.1 représentent les débits obtenus respectivement par Madeleine 2 (1), Madeleine 2 avec un schéma de communication à en-tête (2), et *MadIO* pour $h = 48$ octets (4). Les différences étant imperceptibles, les courbes correspondant à d'autres valeurs de h ne sont pas représentées. Nous observons que pour les messages de plus de 1 ko, la différence entre *MadIO* et Madeleine est infime. Pour les message en dessous de 1 ko, la différence entre *MadIO* et Madeleine est nette. La courbe (2) correspondant à "Madeleine 2 + en-tête", qui coïncide avec les résultats de *MadIO*, indique que ce surcoût est à attribuer presque entièrement à l'en-tête fixe.

Bilan. *MadIO* exploite efficacement Madeleine, aussi bien du point de vue du débit que de la latence. Le débit mis à disposition par Madeleine est utilisé jusqu'à son maximum. À première vue, il peut sembler que les résultats de latence pour *MadIO* soient moins bons que ceux de Madeleine. Cependant, il est préférable de prendre en compte les circonstances dans lesquelles Madeleine et *MadIO* sont utilisés : en utilisation normale — c'est-à-dire avec un exécutif ou un adaptateur qui aurait envoyé de toute façon un en-tête —, le surcoût de l'en-tête de *MadIO* est amorti. Au final, la différence de latence due au multiplexage de *MadIO* est très faible, de valeur moyenne 280 ns et inférieure à 370 ns sur nos *Pentium III* 1 GHz de test.

10.1.2 *NetAccess SysIO*

Le débit obtenu en *ping-pong* par *SysIO* sur TCP/Ethernet-100 est représenté à la figure 10.2. Sur la même figure, nous avons représenté le débit obtenu sur le même réseau en programmant directement TCP/IP par *sockets*. La différence entre le débit obtenu par *SysIO* et directement par *sockets* est inférieure à 2 %.

Sur ce réseau, la latence observée en programmant les *sockets* s'élève à 72 μs. *SysIO* obtient une latence de 77 μs. Une instrumentation du code de *SysIO* a mis en évidence que ce surcoût est dû à la scrutation systématique à l'aide de `select()` ou `poll()` dans *SysIO*. Il serait probablement possible de diminuer ce surcoût en utilisant des méthodes de scrutation spécifiques telles que `kqueue` [126] de BSD et `/dev/epoll` [128] de Linux.

En test *ping-pong*, *SysIO* reste en mode rafale, ce qui explique sa relativement bonne réactivité. Lors de transmissions isolées et que *SysIO* est en mode repos, la latence observée dépend du contexte et des paramètres de régulation de l'équité. Si le message n'est pas attendu, alors le temps d'attente de régulation s'ajoute à la latence — typiquement, +1 ms avec les réglages par défaut. Si le message est

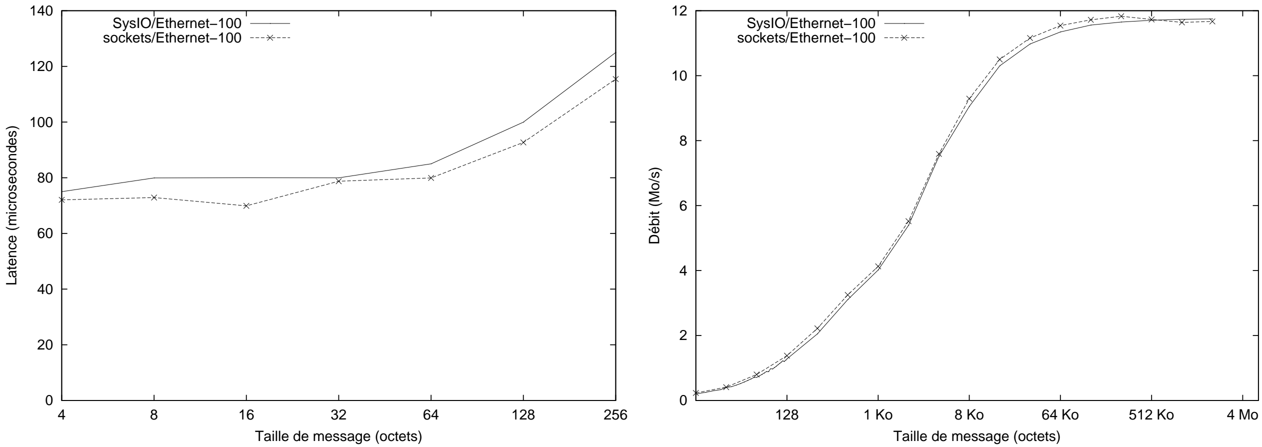

FIG. 10.2 – *Comparaison de* SysIO *et sockets sur Ethernet-100 (à gauche : latence ; à droite : débit).*

attendu et que son arrivée est testée par appel à `sysio_refresh()`, alors nous retrouvons les mêmes performances qu'en mode rafale, soit 77 μs. En pratique, un message attendu est reçu avec la latence de 77 μs ; un message qui n'est pas attendu est reçu avec une latence bien plus élevée si le processus calcule et que *SysIO* n'a pas la main. Dans ce cas, la latence moyenne s'élève à environ 5 ms, temps moyen pour que l'ordonnanceur Marcel donne la main au scrutateur. Le résultat est le même, que *SysIO* utilise `marcel_poll` ou un *thread* dédié prioritaire selon les différentes stratégies décrites en section 8.4 page 111. Ce problème n'est pas spécifique à *SysIO* et affecte tout type de réseaux.

10.1.3 Arbitrage

Nous étudions dans cette section l'efficacité des mécanismes d'arbitrage. Cette étude se décompose en trois parties : l'arbitrage intra-paradigme, l'arbitrage entre *SysIO* et *MadIO*, et l'arbitrage entre la scrutation du réseau et les calculs.

Arbitrage intra-paradigme. Nous avons réalisé l'essentiel de nos tests avec Marcel-mono ; PadicoTM sur Marcel-SMP présentant des problèmes de stabilité, et les travaux sur les activations étant récents, nous nous limitons à la version basée sur Marcel-mono. Ceci n'est guère gênant dans la mesure où seule la version Marcel-mono demande un réel arbitrage intra-paradigme, les autres versions de Marcel fournissant cet arbitrage au niveau de Marcel.

En version "mono", il n'y a pas de réentrance par rapport aux appels systèmes. Les communications par *SysIO* sont donc sérialisées. Lors de communications avec plusieurs pairs simultanément, les communications Madeleine sont réentrantes. Les communications *MadIO* sont donc capables de s'entrelacer. La figure 10.3 représente la latence et le débit obtenus par l'abstraction *VLink*/*MadIO* en client-serveur avec un client (1 vers 1), et avec deux clients concurrents réalisant le même test simultanément avec le même serveur (2 vers 1). La figure représente les mesures relevées sur l'un des deux clients, les deux clients obtenant sensiblement les mêmes performances. Nous constatons une légère pénalité sur la latence dans le cas "2 vers 1". Le débit constaté est conforme aux attentes : le cas "2 vers 1" obtient un maximum à la moitié du cas "1 vers 1".

Arbitrage *MadIO-SysIO*. Les performances de *SysIO* et *MadIO* décrites précédemment ont été mesurées avec les réglages par défaut de *SysIO*, à savoir une attente de régulation de 1 ms quand le scrutateur est au repos, et aucune attente pendant les rafales. En désactivant complètement *SysIO*, aucun modification des performances de *MadIO* n'est observée ; nous concluons que *SysIO* au repos ne rentre pas du tout en concurrence avec *MadIO*. En supprimant l'attente de régulation et en la remplaçant par un simple `yield`, nous observons une latence moyenne de *MadIO* augmentée d'environ

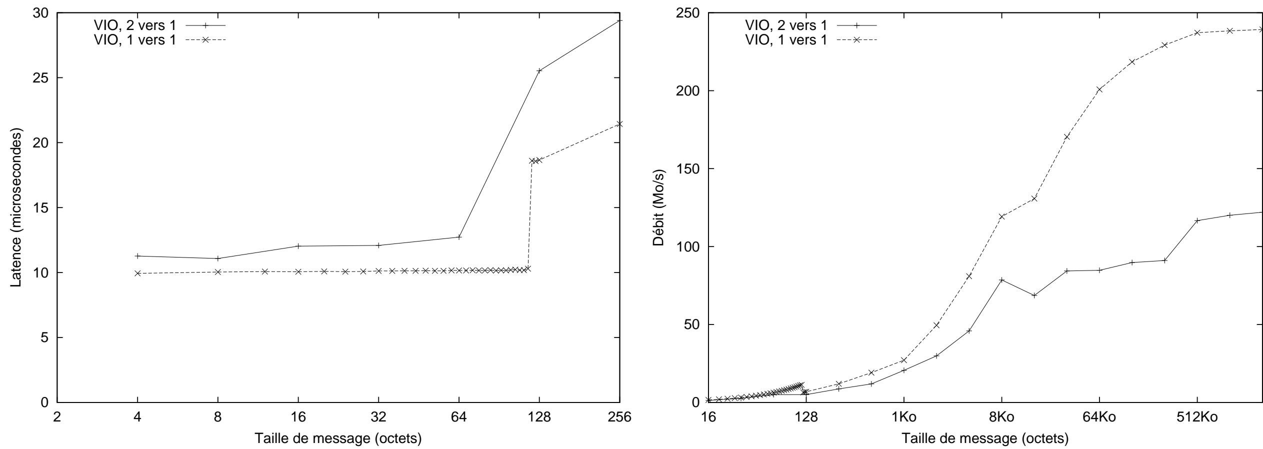

FIG. 10.3 – *Latence et débit mesurés sur* MadIO *avec deux clients simultanés.*

```
gettimeofday(&t1, NULL);
for(i=0; i<iter; i++)
  {
    j++;
  }
gettimeofday(&t2, NULL);
```

FIG. 10.4 – *Programme de test utilisé pour mesurer la concurrence entre scrutation et calculs.*

6 μs, sans affecter le débit maximum. Le seul cas problématique survient si nous supprimons totalement la régulation de *SysIO* et qu'il réalise alors une attente active. Dans ce cas, les performances de *MadIO* deviennent aléatoires, avec une latence moyenne de l'ordre de 10 ms, correspondant au temps moyen pour que l'ordonnanceur Marcel reprenne la main préemptivement sur *SysIO*. Dans le cas général avec une attente de régulation raisonnable dans *SysIO*, les performances de *MadIO* ne sont pas affectées par le scrutateur *SysIO* en tâche de fond.

Arbitrage scrutation-calcul. L'arbitrage doit s'assurer que la scrutation du réseau ne pénalise pas les *threads* de calcul. La scrutation par Madeleine étant intégrée à l'ordonnanceur Marcel, nous n'avons pas beaucoup d'influence sur elle. Nous constatons expérimentalement qu'elle a une influence négligeable.

Pour tester l'influence de la scrutation *SysIO* en tâche de fond, nous réalisons le programme de la figure 10.4 qui réalise un calcul en boucle. Nous réalisons le test sur un *AMD Athlon-XP* 1600+ sous Linux-2.4.21. En environnement normal, nous mesurons 96.4 itérations/μs. Dans PadicoTM, nous mesurons une valeur de 97.9 itérations/μs pour les deux stratégies de *SysIO* avec le *thread* de scrutation en tâche de fond (au repos) et une régulation constituée d'une attente non-nulle ou d'un simple passage de main à l'ordonnanceur Marcel, ou avec la scrutation *SysIO* enregistrée dans `marcel_poll`. L'*accélération* apparente du calcul dans PadicoTM est un artefact de Marcel : son utilisation du signal `SIGALRM` augmente sa priorité pour l'ordonnanceur du noyau ; le processus est donc plus souvent ordonnancé. En mesurant le temps réel et non le temps utilisateur, ceci se traduit par un écart de l'ordre de 1$\sim$2 %, même sur une machine sans autre charge processeur. L'influence réelle de la scrutation dans ce cas semble donc faible. Nous constatons qu'en désactivant complètement le *thread* de *SysIO*, la vitesse de calcul ne change pas. En réalité, la scrutation *SysIO* en elle-même prend environ 5 μs ; si l'on considère qu'avec les deux stratégies considérées, la scrutation est effectuée une fois à chaque tranche de temps de Marcel (10 ms), le surcoût est de 0.05 %, ce qui est négligeable.

Pour une version de *SysIO* construite sur un *thread* dédié sans régulation, nous mesurons 48.6 itérations/μs, ce qui traduit le fait que la scrutation et le *thread* de calcul se partagent le pro-

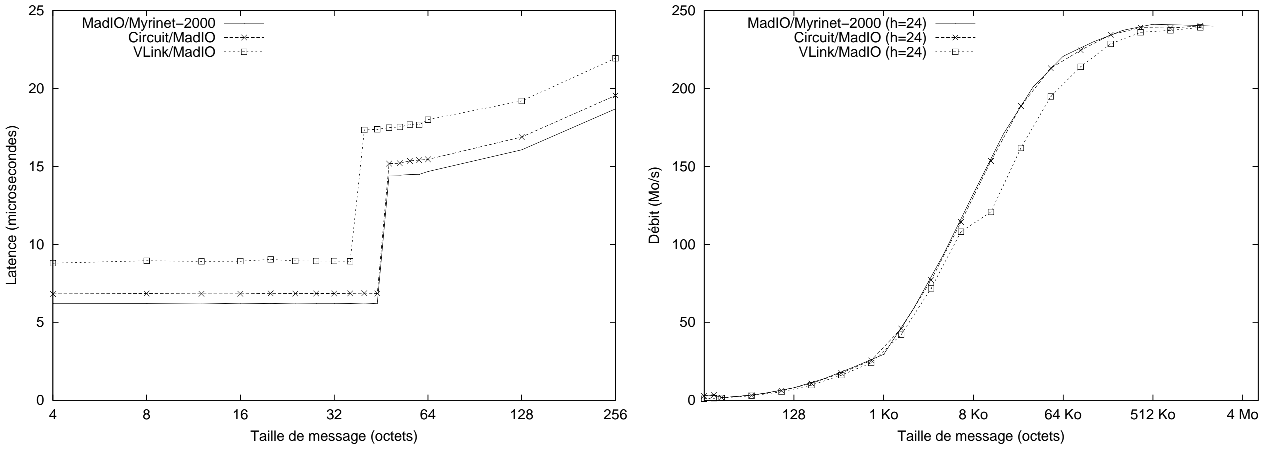

FIG. 10.5 – *Latence et débit des abstractions* Circuit *et* VLink *sur* MadIO/*Myrinet-2000.*

cesseur à égalité. En conclusion, dès que les paramètres de scrutation de *SysIO* sont raisonnables (*ie.* pas d'attente active), le *thread* de scrutation en tâche de fond à une influence quasi-nulle.

10.2 Évaluation des abstractions *VLink* et *Circuit*

Dans cette section, nous évaluons les performances des abstractions construites au-dessus du niveau d'arbitrage. Ces abstractions sont les adaptateurs de la collection principale et leurs personnalités.

Nous avons constaté que pour une abstraction donnée, la différence de performance entre une utilisation directe de l'adaptateur et l'utilisation de l'une des personnalités et infime. Ceci n'est guère étonnant dans la mesure où une personnalité ajoute simplement un niveau d'appel de fonction pour présenter les noms de fonction et l'ordre des paramètres attendus par l'application. Les résultats présentés sont donc indépendants de la personnalité choisie.

La figure 10.5 présente les performances obtenues par les adaptateurs *Circuit* et *VLink* sur *MadIO*. Nous mesurons que le surcoût de latence introduit par la virtualisation de la topologie de *Circuit* s'élève à 400 ns, ce qui reste raisonnable. L'adaptateur *VLink/MadIO* introduit un surcoût de latence de 2 μs. Les débits obtenus par les deux abstractions sur *MadIO* exploitent pleinement le maximum de la bande passante disponible, jusque 240 Mo/s. Le léger décrochement du débit observé pour *VLink/MadIO* aux alentours de 16 ko correspond au seuil pour l'utilisation de *rendez-vous*. L'adaptateur *Circuit* ne présente pas ce décrochement, car les mécanismes de *rendez-vous* sont plutôt mis en œuvre dans les exécutifs l'utilisant et non dans *Circuit*.

Enfin, sur Ethernet-100 par *SysIO*, le surcoût introduit par *Circuit* et *VLink* est imperceptible.

10.3 Performances des exécutifs sur PadicoTM

Nous étudions dans cette section la performances de plusieurs exécutifs au-dessus de PadicoTM. Les performances offertes par les abstractions sur *SysIO* étant très proches des *sockets* standard, nous nous focalisons essentiellement sur le cas de *MadIO* où les exécutifs sont utilisés en dehors de leur contexte habituel.

10.3.1 MPI

Nous avons évalué les performances de MPICH-padico avec les métriques habituelles, à savoir la latence et le débit. La figure 10.6 présente les performances comparées de MPICH-padico pour diverses valeurs de h, et de MPICH/*Madeleine* sur Myrinet-2000. MPICH/*Madeleine* est une implémen-

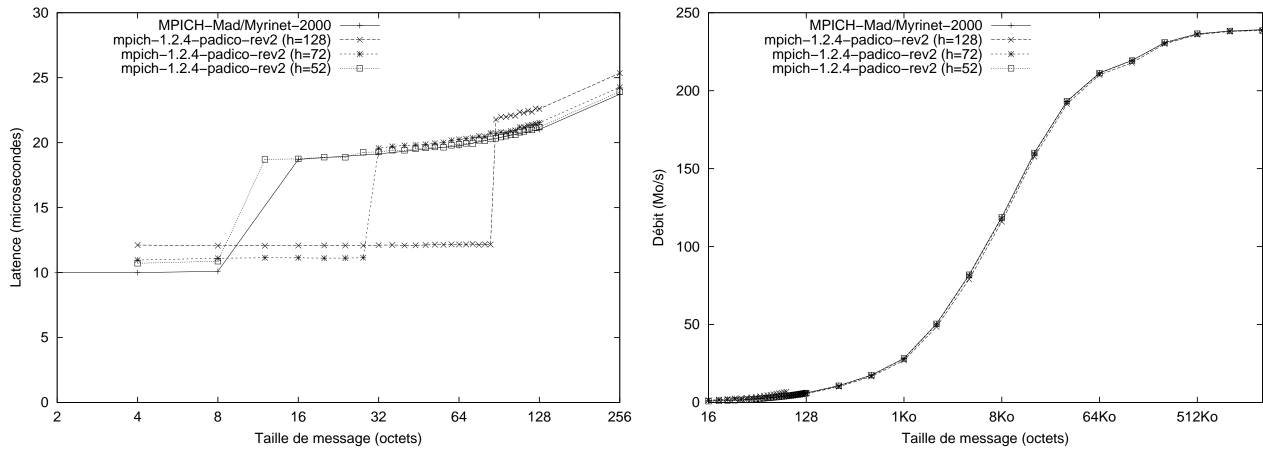

FIG. 10.6 – *Latence et débit de* MPICH*-padico et* MPICH/Madeleine *sur Myrinet-2000.*

tation MPI, variante de MPICH, construite au-dessus de Madeleine ; MPICH-padico est en une version modifiée pour utiliser l'interface *Circuit* à la place de Madeleine. La comparaison de MPICH/*Madeleine* et MPICH-padico permet donc d'évaluer le surcoût introduit par *MadIO* et *Circuit* en utilisation par un exécutif.

Le profil de la courbe représentant le temps de transfert de MPICH-padico présente les mêmes caractéristiques que *Circuit*/*MadIO* : un pallier constant pour des tailles de données inférieure à la taille pouvant être contenue dans l'en-tête, puis un saut suivi d'une croissance régulière. MPICH/*Madeleine* transporte également des données dans les en-têtes, et présente donc le même type de profil de temps de transfert. Par défaut, MPICH/*Madeleine* transporte 8 octets de données dans ses en-têtes, pour une taille totale d'en-tête de 48 octets. Ce cas correspond au réglage $h = 52$ de *MadIO*. Pour ce réglage, les latences de MPICH/*Madeleine* et MPICH-padico s'élèvent respectivement à 9.93 μs et 10.71 μs. La différence de 780 ns est à attribuer à : 80 ns pour les 4 octets supplémentaires transmis sur le réseau par *MadIO*, 300 ns pour le multiplexage de *MadIO*, et 400 ns pour la virtualisation de topologie de *Circuit*. Pour $h = 128$, la latence s'élève à seulement 12.06 ns et est constante jusque des messages de 84 octets — taille équivalente à 10 flottants en double précision.

La partie droite de la figure 10.6 représente le débit de MPICH-padico pour diverses valeurs du paramètre h de *MadIO*, et le débit obtenu par MPICH/*Madeleine*, sur Myrinet-2000. L'influence de la taille d'en-tête et le surcoût inférieur à la microseconde font que les différences de débit sont imperceptibles au-delà de 128 octets.

Sur les réseaux Ethernet utilisés par TCP/IP, le comportement de MPICH-padico et MPICH/*Madeleine* est tout-à-fait comparable. Il est plus difficile que sur Myrinet de chiffrer le surcoût précisément en raison d'expériences moins facilement reproductibles ; autant que la précision des mesures le permet, nous n'avons décelé aucune différence de performance entre MPICH-padico et MPICH/*Madeleine* sur Ethernet-100.

En conclusion, MPICH-padico obtient des performances très similaires à MPICH/*Madeleine*, implémentation dont elle dérive. Le surcoût de PadicoTM se limite à 780 ns sur la latence et une influence imperceptible sur le débit.

10.3.2 CORBA

Nous présentons les performances obtenues par les implémentations CORBA suivantes : OmniORB (versions 3.0.2 et 4.0.2), MICO 2.3.8, et ORBacus 4.0.5, basées sur des *sockets*, et utilisant la personnalité *SysW* de *VLink*.

Caractérisation des performances de CORBA. L'OMG suggère une méthodologie [145] de mesure des performances des implémentations CORBA, et de l'ORB en particulier. Nous proposons un bench-

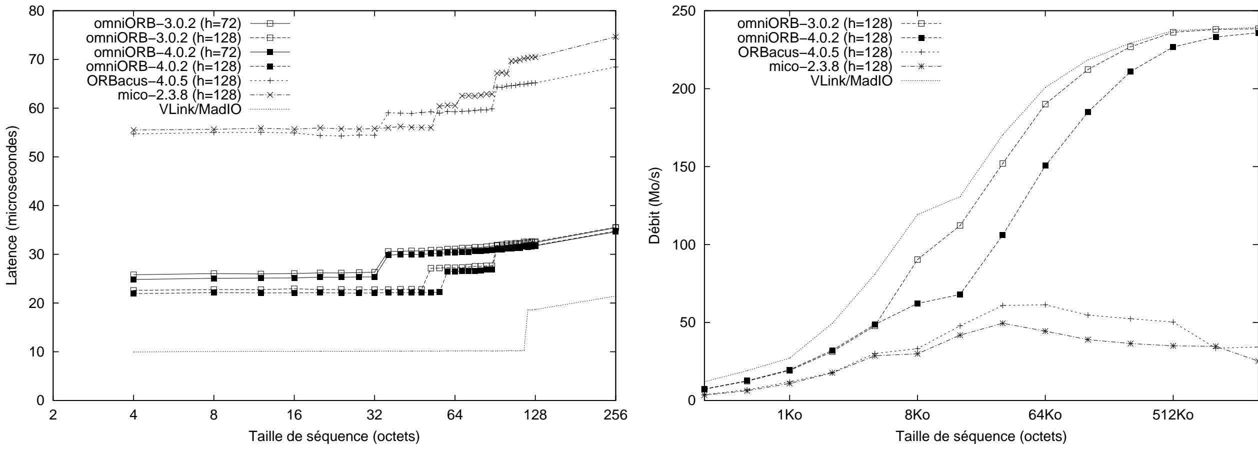

FIG. 10.7 – *Latence et débit de diverses implémentations* CORBA *sur Myrinet-2000.*

mark qui respecte les principes de base donnés par l'OMG, à savoir : adéquation avec l'utilisation typique, portabilité, extensibilité et simplicité. L'utilisation que nous visons pour CORBA est le calcul haute performance. Nous ne porterons donc pas notre attention aux performances des divers services de CORBA qui ne sont pas utilisés dans ce cadre. Nous nous concentrerons plutôt sur le cœur de l'ORB : l'invocation d'une méthode distante. Puisque nous serons amenés à comparer CORBA à MPI et d'autres exécutifs, il est naturel d'utiliser les mêmes métriques que celles utilisées pour évaluer MPI plutôt que la métrique habituelle dans le monde CORBA (nombre de requêtes par seconde). Nous retenons donc comme grandeurs caractéristiques la latence et le débit que nous définissons ci-dessous.

Une invocation CORBA se décompose en les étapes suivantes :

1. invocation de la souche dans le processus client, encodage des arguments ;
2. émission de la requête sur le réseau — elle contient : un en-tête GIOP (structure fixe, 12 octets), une description de requête (taille variable), et les arguments.
3. invocation du squelette dans le processus serveur, décodage des arguments ;
4. exécution de la méthode ;
5. retour au squelette, encodage des valeurs de retour ;
6. émission de la réponse sur le réseau — elle contient : un en-tête GIOP (12 octets), une notification de fin de requête (12 octets), et éventuellement les valeurs de retour ;
7. retour dans la souche de l'appelant, décodage des valeurs de retour.

Seules les étapes 2 et 6 sont changées par PadicoTM. Nous définissons alors le temps d'aller-retour comme le temps d'invocation d'une méthode vide sans argument ni valeur de retour, et la *latence* comme la moitié du temps d'aller retour. Nous calculons le *débit* à partir du temps d'invocation d'une méthode qui prend en argument `inout` (transmis en paramètre et valeur de retour) une séquence d'entiers de longueur variable.

La figure 10.7 présente les performances de plusieurs implémentations CORBA dans PadicoTM sur le réseau Myrinet-2000. Les courbes représentent le temps de transfert et le débit en fonction de la taille des données transmises. Les performances brutes de *VLink/MadIO* sont rappelées à titre de référence. Les performances obtenues par ces exécutifs sur Ethernet-100 par *SysIO* sont semblables aux performances en dehors de PadicoTM directement sur TCP/IP.

Latence et petits messages. La partie gauche de la figure 10.7 représente le temps de transfert pour les petites tailles de données. Ces courbes présentent des paliers correspondant aux paliers de *VLink/MadIO* selon que l'envoi nécessite deux paquets ou que toutes les données sont transportées dans l'en-tête en un seul paquet, comme mis en évidence à la figure 10.5. Le message "aller" de notre test contient, outre les données et les 12 octets GIOP, une description de requête de 48 octets ; avec les en-têtes *MadIO* et *VLink*, le total d'en-têtes est de 72 octets, d'où le saut observé pour les séquences de

taille 56 octets quand $h = 128$. Le message "retour" contient 12 octets d'en-tête, une notification de 12 octets, et les données, soit un total d'en-tête de 36 octets, d'où le saut observé pour les séquences de 96 octets quand $h = 128$.

La latence mesurée pour une invocation de méthode sans argument est de 18.4 μs pour OmniORB 4 et 21 μs pour OmniORB 3. ORBacus et MICO obtiennent respectivement 51.6 μs et 50.1 μs. Les meilleurs résultats de latence sont obtenus pour des valeurs de h calculées au plus juste — typiquement, 80 octets. Ces chiffres sont majorés d'une microseconde pour $h = 128$. Les médiocres performances de MICO et ORBacus s'expliquent par des surcoûts logiciels (étapes 1, 3, 5 et 7) importants.

Débit. La partie droite de la figure 10.7 représente le débit mesuré de ces mêmes implémentations CORBA sur Myrinet-2000. Le paramètre h n'ayant pas d'influence significative sur le débit, nous ne représentons que le cas $h = 128$. Les deux versions d'OmniORB exploitent la bande passante disponible jusqu'à son maximum à 240 Mo/s. En revanche, MICO et ORBacus ne parviennent pas à exploiter correctement la bande passante, obtenant respectivement 50 et 61 Mo/s. À la différence d'OmniORB, ces deux ORB effectuent des copies en mémoire pour l'encodage et le décodage des arguments.

L'encodage, l'émission sur le réseau, et le décodage étant réalisés successivement, le temps total $T(l)$ de l'étape 1 à l'étape 4 peut s'écrire :

$$T(l) = T_{\text{encodage}} + T_{\text{réseau}} + T_{\text{décodage}} \tag{10.2}$$

avec : $T_{\text{encodage}} = T_{\text{décodage}} = \beta_{\text{mémoire}} + \frac{l}{D_{\text{copie mémoire}}}$
et $T_{\text{réseau}} = \beta + l\tau$

En considérant le comportement asymptotique pour de grandes valeurs de l, nous pouvons négliger les latences pures (β) de ces termes. En posant $D_{\text{réseau}} = \frac{1}{\tau}$, le débit global asymptotique D est caractérisé par :

$$\frac{1}{D} = \frac{1}{D_{\text{réseau}}} + \frac{2}{D_{\text{copie mémoire}}} \tag{10.3}$$

Sur nos machines de test *Pentium III* 1 GHz, nous mesurons le débit de copie de mémoire à mémoire $D_{\text{copie mémoire}} = 230$ Mo/s. Pour le réseau Myrinet-2000 utilisé par VIO, $D_{\text{réseau}} = 240$ Mo/s. Le débit maximum théorique s'élève alors à $D = 77$ Mo/s ; MICO et ORBacus n'atteignent respectivement que 50 et 61 Mo/s, principalement à cause de leur encodage qui encode les séquences élément par élément au lieu d'une simple copie. Pour ces ORB, en réalité $D_{\text{encodage}} < D_{\text{copie mémoire}}$. En instrumentant le code, nous avons mesuré que la vitesse d'encodage réelle de MICO plafonne à 140 Mo/s et le décodage à 170 Mo/s. Ceci donne un débit global de 58 Mo/s, relativement proche des valeurs expérimentales.

Nous concluons que même en améliorant l'encodage de MICO ou ORBacus, le débit ne dépassera jamais 77 Mo/s sur Myrinet-2000 tant qu'ils procéderont à des copies des argument en mémoire. D'une manière générale, les débits des réseaux haut débit et de la mémoire progressant ensemble et restant du même ordre de grandeur, un ORB qui encode et décode en faisant une copie ne peut pas dépasser environ 30 % du débit nominal du réseau. OmniORB, qui réalise l'encodage la plupart du temps sans effectuer de copie en environnement homogène, ne présente pas cette limitation.

10.3.3 Autres exécutifs

gSOAP. Nous avons réalisé une mesure de performances de l'implémentation gSOAP dans PadicoTM. Sur Myrinet-2000, la latence s'élève à environ 650 μs et le débit ne dépasse pas 3.3 Mo/s pour les chaînes de caractères et 1 Mo/s pour les types structurés ; sur Ethernet-100 sans PadicoTM, la latence est de 1.7 ms pour un débit identique. Le goulot d'étranglement étant l'encodage/décodage XML et non le réseau, le passage par Myrinet ne change pas grand chose au coût global.

Java. Les communications par *socket* (*via* VIO) de la version de la machine virtuelle Java Kaffe dans PadicoTM obtient sur Ethernet-100 avec *SysIO* des performances semblables à la version sans

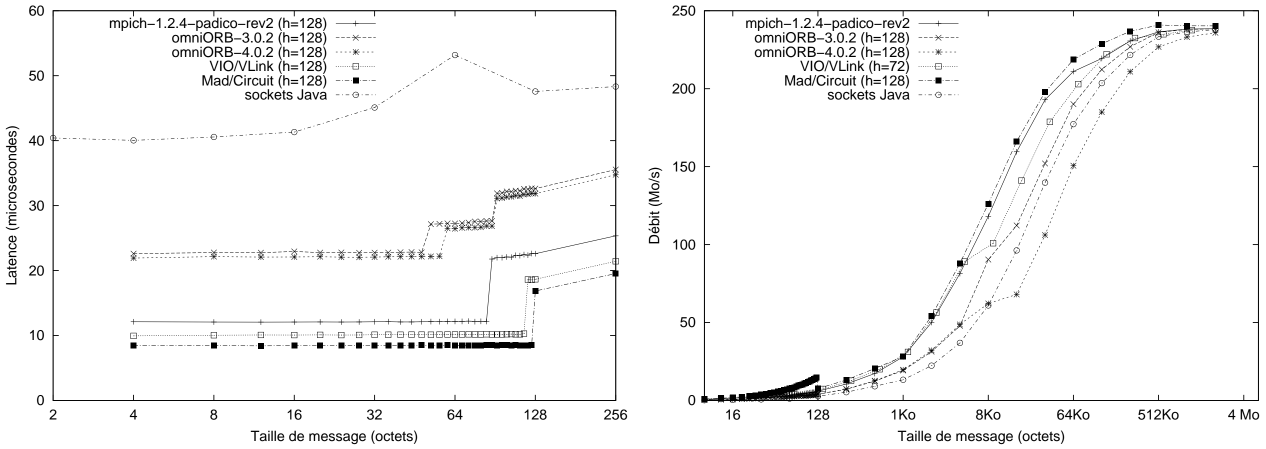

FIG. 10.8 – *Comparaison des performances de divers exécutifs et interfaces sur PadicoTM.*

Exécutif/interface	*Circuit*	*VLink*	MPICH-1.2.5	OmniORB 3	OmniORB 4	*sockets* Java
Latence (μs)	8.4	10.2	12.06	20.3	18.4	40
Débit maximum (Mo/s)	240	239	238.7	238.4	235.8	237.9
Demie bande passante (taille de message, octets)	7562	9616	8997	18850	41604	25444

TAB. 10.1 – *Récapitulatif des performances de divers exécutifs pour $h = 128$ sur* MadIO/*Myrinet-2000.*

PadicoTM. Lors de l'utilisation de Myrinet-2000, la latence au niveau *socket* Java s'élève à 40 μs. Le débit correspond à bonne une utilisation de VIO. Le surcoût de latence est élevé ; Kaffe n'est pas réputée être la machine virtuelle Java la plus rapide. Cependant, le code source des machines virtuelles Java les plus performantes n'est pas libre. Nous envisageons à l'avenir de nous pencher sur `gcj`, le compilateur Java natif du projet `gcc`, qui semble prometteur. Nous envisageons également d'étudier la réutilisation de code binaire après édition de lien — et pas seulement avant édition de lien, comme actuellement dans PadicoTM — pour pouvoir utiliser les implémentations qui ne sont pas libres et fournies uniquement sous forme binaire liée.

10.3.4 Comparaison et discussion

La figure 10.8 présente une comparaison des performances des exécutifs dans PadicoTM sur Myrinet-2000, ainsi que de la personnalité VIO sur *VLink* et Madeleine sur *Circuit* qui peuvent servir de base pour d'autres exécutifs dans PadicoTM. MICO, ORBacus, et gSOAP aux performances médiocres, ne sont pas représentés. Nous avons choisi une unique valeur de h pour ce comparatif, de façon à comparer les performances des exécutifs dans les mêmes conditions. Nous avons choisi = 128 ; c'est le choix qui donne le meilleur compromis entre MPI et CORBA. En effet, les contraintes sont fortes pour trouver la valeur à utiliser. Une session PadicoTM doit utiliser la même valeur pour tous les exécutifs, sur tous les nœuds. MPICH-padico demande au moins 44 octets ; la latence CORBA est très élevée pour $h < 72$. La différence de latence entre $h = 72$ et $h = 128$ étant très faible et $h = 128$ apportant des performances intéressantes pour les petits messages, nous choisissons $h = 128$ pour nos tests.

Le tableau 10.1 récapitule les performances obtenues par les exécutifs et abstractions dans PadicoTM, sous la forme d'une latence qui caractérise les petits messages, d'un débit qui caractérise les gros messages, et de la taille de message pour laquelle le débit obtenu est la moitié du maximum, ce qui caractérise la performance pour des messages de taille moyenne.

Tous ces exécutifs réussissent à exploiter le maximum du débit mis à leur disposition, avec des maxima compris entre 235 et 240 Mo/s. Cependant, la croissance du débit en fonction de la taille des messages n'est pas la même pour tous les exécutifs. MPICH-padico est celui à la croissance de débit la plus rapide, avec très peu d'écart part rapport à *Circuit*. À l'inverse, OmniORB 4 se fait distancer

Kaffe 30	omniORB 8.2 – 9.7	MPICH 3.7 – 4.2
VLink 2 – 2.5		Circuit 0.4 – 0.5
NetAccess: SysIO 4 – 6		MadIO 0.25
sockets TCP/IP Ethernet–100 72	Madeleine Myrinet–2000 5.5 – 7.8	

FIG. 10.9 – *Coût logiciel et matériel estimé de chaque module lors de l'envoi de messages ; les temps sont donnés en microsecondes, sous la forme d'un intervalle de valeurs constatées selon les conditions. Ces temps comprennent le coût d'émission et de réception additionnés.*

pour des tailles de messages entre 8 et 64 ko ; la rupture dans la courbe de débit à 16 ko correspond à un changement de sa stratégie d'empaquetage pour éviter les copies. Là où MPICH-padico atteint sa demie bande passante (soit 120 Mo/s) pour des messages d'environ 8 ko, OmniORB 4 n'y parvient qu'à plus de 40 ko.

Pour les petits messages, le choix du paramètre $h = 128$ assure des performances correctes aussi bien à MPICH-padico qu'à OmniORB. La latence MPI n'est alors certes pas le minimum possible, mais elle est constante pour des messages jusque 84 octets. Cet en-tête suffisamment grand assure une bonne agrégation des en-têtes CORBA, *VLink* et *MadIO*, pour invocations simples transportées en un seul paquet. Nous obtenons ainsi une latence CORBA à peine 50 % plus élevée que la latence MPI, et un débit maximum sensiblement égal. PadicoTM permet donc à cette implémentation CORBA d'afficher des performances proches de MPI, y compris sur les réseaux haut débit de type Myrinet. De plus, ces résultats sont obtenus pour des implémentations CORBA sans modification, c'est-à-dire qui utilisent le protocole IIOP. Il serait sans doute possible d'améliorer encore ces performances en modifiant l'implémentation pour qu'elle utilise directement les services de PadicoTM, sans passer par le protocole standard CORBA IIOP et la virtualisation. Ce type de modification, appelée ESIOP (*Environment Specific Inter-ORB Protocol*), est prévu par la norme CORBA.

La figure 10.9 illustre l'empilement des couches logicielles et résume le coût introduit par chaque module (les valeurs sont données en microsecondes).

10.4 Méthodes de communication pour WAN

10.4.1 ParallelStreams

Nous avons évalué les performances de *ParallelStreams* sur le réseau VTHD. Le réseau VTHD est un réseau expérimental français à grande échelle. Il est caractérisé par un haut débit (2.5 Gbit/s, soit 312 Mo/s) et, en raison des longues distances, une latence élevée. Nous avons utilisé le lien Rennes-Nice, soit environ 1000 km. La latence typique de ce lien est mesurée à 8 ms. Les machines sont reliées au réseau VTHD par Ethernet-100 ; chaque machine peut donc accéder à VTHD à 100 Mbit/s au maximum.

Le débit obtenu avec et sans *ParallelStreams* sur ce lien est représenté à la figure 10.10 en fonction de la taille de message. Dans les deux cas, la courbe est irrégulière, avec certaines tailles de messages présentant des performances pathologiques (décrochement à 256 ko, par exemple). Les *sockets* avec et sans *ParallelStreams* utilisent les même réglages qui, d'après nos expérimentations, obtiennent les meilleures performances que nous avons pu obtenir sur ce lien. Ces réglages sont d'une part les tailles de fenêtre TCP élargies au maximum (256 ko sur nos machines de test), et d'autre part la désactivation de l'algorithme de Nagle à l'aide du drapeau `TCP_NODELAY`, de façon à contrôler finement l'envoi des paquets sur le réseau. Les paramètres de *ParallelStreams* sont : 3 flux, avec un entrelacement entre

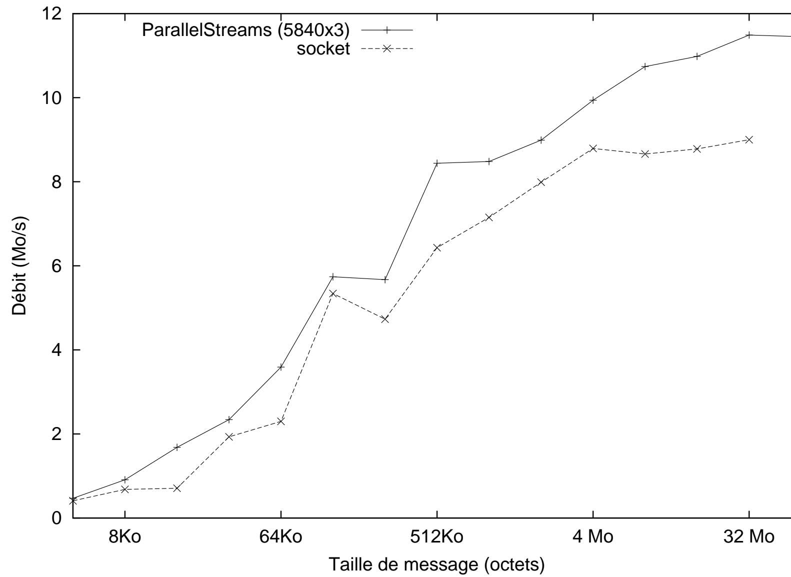

FIG. 10.10 – *Débit mesuré sur* VTHD *avec des flux parallèles.*

version de AdOC	fichier *pdf*	fichier *ps*
AdOC-0.5 standard	694 kbit/s (1.31)	1140 kbit/s (2.71)
VLink/AdOC-0.5 dans PadicoTM	720 kbit/s (1.32)	1360 kbit/s (2.84)

TAB. 10.2 – *Débit apparent et taux de compression effectif avec AdOC sur une ligne* ADSL *à 608 kbit/s.*

les flux basé sur des blocs de 5840 octets (équivalent à la charge utile au niveau TCP/IP de 4 trames Ethernet).

L'effet constaté des *ParallelStreams* est un lissage de la courbe de débit d'une part, et l'amélioration du débit global d'autre part. Toutefois, la latence élevée (8 ms) fait que le débit maximum est obtenu pour de très grands messages de 32 Mo, alors que ce maximum est déjà atteint pour des messages de 64 ko sur un réseau Ethernet-100 local.

Les performances obtenues en CORBA ou MPI au-dessus des *ParallelStreams* sont similaires aux performances brutes de *ParallelStreams*. Pour cet ordre de grandeur de latence et débit du réseau, le coût du réseau domine largement le coût logiciel.

10.4.2 AdOC

Pour évaluer les performance de la compression par AdOC, nous avons utilisé une ligne ADSL à 608 kbit/s. Avec cet ordre de grandeur de débit, il devient rentable de compresser les données à la volée.

Le tableau 10.2 présente le débit mesuré lors de cette expérience sur ligne à 608 kbit/s. La compression étant largement dépendante des données envoyées, nous avons utilisé pour ce test deux fichiers au comportement différent vis-à-vis de la compression : un fichier `ps` composé de texte susceptible d'être compressé, et un fichier `pdf` qui utilise déjà la compression en interne, que nous voyons comme un fichier binaire difficile à compresser. Le tableau 10.2 présente les débits constatés au niveau applicatif ainsi que le taux de compression effectif qui a été utilisé par AdOC lors de la transmission. L'utilisation de la compression permet un débit apparent bien au-delà du débit nominal de la ligne, avec une meilleure compression sur le fichier texte que sur le fichier binaire. Les performances constatées pour *VLink*/AdOC dans PadicoTM sont du même ordre de grandeur que celles constatées pour AdOC seul ; la version PadicoTM est même légèrement plus rapide. Nous attribuons cette légère différence à l'utilisation de *threads* Marcel dans PadicoTM : l'utilisation du signal `SIGALRM` par Marcel

augmente la priorité du processus pour l'ordonnanceur système ; le temps processeur gagné ainsi permet à AdOC de mieux compresser les données. Il s'agit du même effet que celui constaté lors des mesures de l'arbitrage calcul-communications.

Nous n'avons pas observé de différence notable entre les performances brutes obtenues par *VLink*/AdOC et les performances obtenues avec un exécutif qui utilise *VLink*/AdOC. Le coût induit par le réseau domine largement les surcoûts logiciels des exécutif.

PadicoTM tire donc pleinement profit de AdOC et met la compression à disposition de tous les exécutifs qu'il supporte.

10.4.3 VRP

Nous avons évalué les performances du protocole à tolérance de pertes VRP en nous basant sur une implémentation préliminaire. Sur des liens longue distance lents, autoriser quelques pertes permet une amélioration appréciable. Par exemple, une expérience menée en 2000 sur le lien Chicago-Los Angeles montre que TCP réalise des transferts à environ 1.2 Mbit/s ; en autorisant 20 % de pertes avec VRP, le débit constaté est de 6 Mbit/s, et nous mesurons un taux de pertes effectif d'environ 10 %. D'autres améliorations spectaculaires ont été constatées sur des liens transcontinentaux.

Les travaux autour du protocole VRP seront continués dans le cadre de l'action conjointe ARC-ACI "ALTA" décrite au chapitre suivant.

10.5 Conclusion

Nous avons présenté dans ce chapitre une évaluation des performances de PadicoTM et des exécutifs dans PadicoTM. Dans le cas de combinaisons exécutifs/réseau permises par les implémentations originales des exécutifs, nous avons de plus comparé les performances dans PadicoTM avec les implémentations originales, de façon à évaluer surcoût introduit par PadicoTM.

Nous avons mis en évidence que pour les combinaisons exécutif/réseau permises par l'implémentation originale de l'exécutif, le surcoût introduit par PadicoTM est négligeable ; c'est notamment le cas de CORBA/TCP (surcoût imperceptible) et de MPICH/Myrinet (surcoût inférieur à une microseconde). Nous avons également évalué les performances d'exécutifs sur des réseaux pour lesquels ils ne sont pas conçus, et dont l'utilisation est rendue possible par PadicoTM. C'est principalement le cas des implémentations CORBA sur les réseaux haut débit. Les résultats sont contrastés : nous observons d'une part les implémentations qui se soucient de la performance (OmniORB) et qui obtiennent de bonnes performances, et d'autre part les implémentations qui réalisent des copies en mémoire (MICO, ORBacus) et qui obtiennent des performances médiocres.

Nous avons mesuré les performances de méthodes spécifiques aux WAN. Les résultats sont concluants ; les différentes méthodes — *ParallelStreams*, compression et tolérance de perte — permettent effectivement d'améliorer la performances dans les cas pour lesquels elles sont prévues.

PadicoTM, en tant que carrefour entre les réseaux et les exécutifs, permet donc à chaque exécutif d'utiliser efficacement chaque type de réseau. Le surcoût propre de PadicoTM par rapport à une implémentation spécifique de chaque exécutif sur chaque type de réseau ou chaque méthode de communication est négligeable.

Chapitre 11

Utilisations de PadicoTM

Sommaire

11.1 CORBA parallèle : les environnements PaCO++ et GRIDCCM **159**
11.2 Projet GRID-RMI **160**
11.3 Projet HydroGRID **160**
11.4 Projet EPSN **161**
11.5 L'environnement DIET et le projet GASP **162**
11.6 Projet ALTA **162**
11.7 Conclusion **163**

Dans ce chapitre, nous présentons les projets qui utilisent PadicoTM. La programmation des grilles met de plus en plus souvent en jeu plusieurs exécutifs pour différentes tâches parmi le parallélisme, le contrôle, la visualisation et le couplage de codes. De plus, les applications pour les grilles sont amenées à être déployées sur des ressources variée. Il est donc nécessaire pour ce type d'application de pouvoir exploiter plusieurs exécutifs en même temps, sur plusieurs réseaux en même temps ; dans tous les cas, il est souhaitable de pouvoir utiliser la méthode de communication la plus appropriée au réseau.

C'est dans ce but que divers projets utilisent PadicoTM. Ainsi, PadicoTM assure les communications des environnements PaCO++ et GRIDCCM. PadicoTM est également la plate-forme de communication pour les projets GRID-RMI, EPSN et DIET. Enfin, PadicoTM sert de support pour les communications dans le projet ALTA [21]. Nous décrivons dans ce chapitre ces projets et ce que leur apporte l'utilisation de PadicoTM.

11.1 CORBA parallèle : les environnements PaCO++ et GRIDCCM

Les environnements PaCO++ [153] et GRIDCCM [152] sont des extensions respectivement des modèles d'objets et de composants CORBA vers le parallélisme. Ces deux environnements sont développés dans le projet PARIS à l'IRISA.

Le principe de PaCO++ et GRIDCCM est d'encapsuler un code parallèle dans un objet (pour PaCO++) ou un composant CORBA (pour GRIDCCM). Là où une approche basique cache les nœuds multiples de l'objet ou du composant derrière un nœud "passerelle", l'approche retenue dans PaCO++ et GRIDCCM est de connecter tous les nœuds par CORBA de façon à permettre des transferts en parallèle lors de communication d'objet parallèle à objet parallèle. Cette approche est illustré par la figure 11.1. De plus, lors d'une communication entre entités parallèles, l'environnement PaCO++ ou GRIDCCM assure la redistribution des données à la volée si la distribution diffère entre client et serveur.

Dans l'exemple de composants parallèles basés sur MPI, comme représenté à la figure 11.1, dans chaque nœud le code applicatif utilise CORBA et MPI, et l'environnement GRIDCCM lui-même utilise

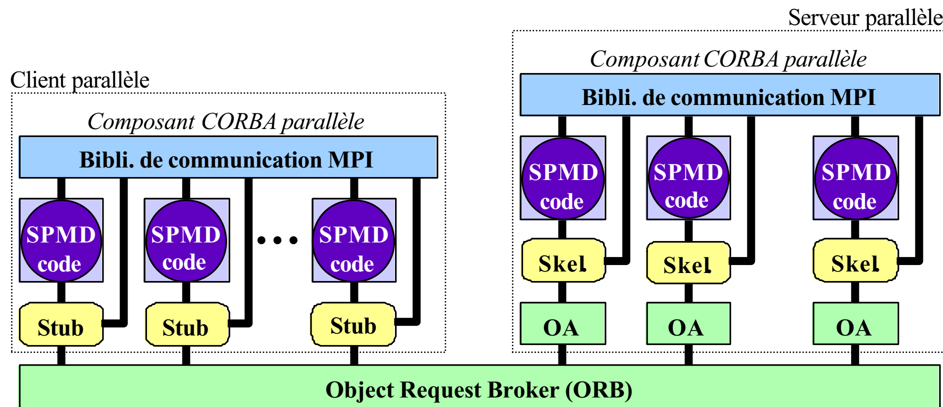

FIG. 11.1 – *Exemple d'utilisation de* GRIDCCM.

CORBA et MPI. PadicoTM apporte alors la possibilité d'utiliser ces divers exécutifs en même temps sur les mêmes ressources, et leur amène la possibilité d'utiliser des ressources et méthodes supplémentaires. Par exemple, quand tous les nœuds sont déployés sur la même grappe, PadicoTM permet aux communications CORBA d'emprunter le réseau haute performance de la grappe.

11.2 Projet GRID-RMI

Le projet connu sous le nom "GRID-RMI" est le projet logiciel "RMI" de l'action concertée incitative GRID 2001 du ministère de la recherche. Il est intitulé « *Objets distribués haute performance pour la grille de calcul* ». Les partenaires de GRID-RMI sont le projet PARIS de l'IRISA, le projet REMAP du LIP, l'équipe GOAL du LIFL, le projet OASIS de l'INRIA Sophia Antipolis, et EADS.

L'objectif principal du projet GRID-RMI est de promouvoir un modèle de programmation pour les grilles de calcul combinant à la fois des modèles du calcul parallèle et du calcul réparti. Les aspects liés au calcul réparti s'appuient sur le concept d'objets distribués — CORBA et Java — et de composants — modèle de composant de CORBA. Le parallélisme prend en compte aussi bien l'échange de message par MPI que la mémoire partagée avec la DSM MOME.

Un tel modèle n'est viable que si l'on est capable d'exploiter l'ensemble des ressources offertes par une grille de calcul, et en particulier les réseaux d'interconnexion. Une application doit être capable d'exploiter les réseaux longue distance où les réseaux haute performance de façon transparente. Le deuxième objectif de ce projet est de concevoir une plate-forme d'objets distribués qui prend en compte des ressources réseaux variées afin d'être capable d'exploiter, de la façon la plus transparente possible, les performances des réseaux sous-jacents.

La figure 11.2 représente l'architecture logicielle mise en œuvre par le projet GRID-RMI. Les applications reposent chacune sur un ou plusieurs environnements parmi : PaCO++, OpenCCM, Do! et PROACTIVE. Les environnements reposent eux-mêmes chacun sur un ou plusieurs exécutifs parmi MPI, CORBA, DSM et JVM. Ces exécutifs se reposent sur PadicoTM qui leur permet d'utiliser les différentes ressources réseau.

PadicoTM apparaît donc comme le cœur du projet GRID-RMI. Il sert de fondation à chaque environnement et chaque exécutif utilisé dans le projet. Du point de vue de PadicoTM, les quatre environnements PaCO++, OpenCCM, *Do!* et PROACTIVE sont vus comme des utilisateurs des exécutifs offerts par PadicoTM, à savoir MPICH, les implémentation CORBA OmniORB et MICOCCM, la JVM Kaffe, et MOME.

11.3 Projet HydroGRID

HydroGRID est un projet pluridisciplinaire de l'Action Concertée Incitative GRID 2002 du ministère de la recherche. Il est intitulé « *Couplage de codes pour le transfert de fluides et de solutés dans les milieux*

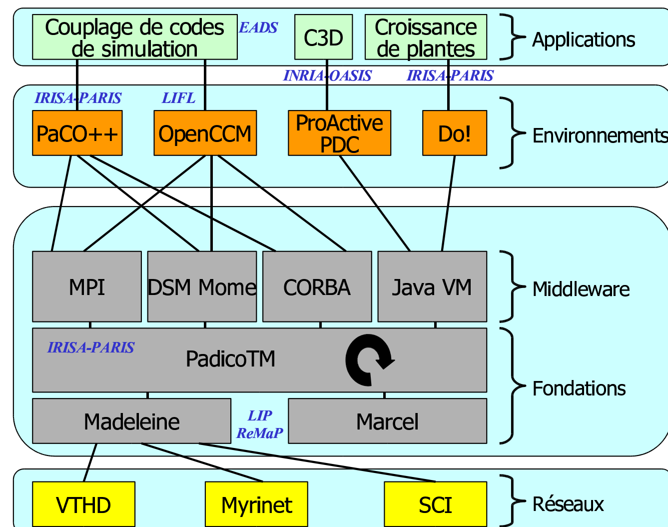

FIG. 11.2 – *Architecture logicielle pour le projet* GRID-RMI.

géologiques : une approche par composants logiciels ». Les partenaires impliqués sont les projets ALADIN et PARIS de l'IRISA, le projet ESTIME de l'INRIA Rocquencourt, l'équipe Hydrodynamique et Transferts en Milieux Poreux de l'IMFS Strasbourg, et l'équipe Transferts Physiques et Chimiques du laboratoire Géosciences de Rennes.

Le projet HydroGRID a pour but de modéliser et simuler des transferts de fluides et de transport de solutés dans des milieux géologiques souterrains. Des problèmes apparus récemment, comme la contamination des aquifères par des polluants, l'intrusion d'eau salée dans les aquifères, et le stockage profond des déchets nucléaires, ont fait apparaître l'importance de phénomènes physico-chimiques complexes. Chaque phénomène physico-chimique est simulé par un code spécifique. Pour simuler numériquement un problème physique couplé, il est souhaitable de laisser ces codes intacts : les modèles sont discrétisés par des méthodes différentes selon les propriétés physico-chimiques, certains logiciels peuvent nécessiter des bibliothèques numériques qui ne sont disponibles que dans certains centres de calcul, etc. Afin de pouvoir interfacer simplement et efficacement ces différents codes, le choix qui a été fait dans HydroGRID est le couplage : chaque code est un composant logiciel de l'application qui n'interagit avec les autres codes qu'au niveau de son interface.

L'intérêt d'utiliser PadicoTM dans HydroGRID découle directement de l'utilisation de composants parallèles. En effet, les différents codes couplés encapsulés dans des composants sont pour la plupart parallèles, développés avec des méthodes qui leur sont propres. Les codes utilisent donc à la fois un exécutif pour le couplage — le modèle de composant CORBA —, et un exécutif pour le parallélisme — typiquement MPI. PadicoTM apporte donc la possibilité d'utiliser ces exécutifs en même temps, et donc rend possible d'encapsuler des codes parallèles dans des composants. D'autre part, PadicoTM permet à tous ces exécutifs un accès égal aux réseaux, c'est-à-dire aussi bien MPI que CORBA peuvent utiliser les SAN, les LAN et les WAN.

11.4 Projet EPSN

Le projet "EPSN" est un projet logiciel de l'Action Concertée Incitative GRID 2001 du ministère de la recherche. Il est intitulé « *Environnement pour le Pilotage de Simulations Numériques distribuées* ». Les partenaires impliqués sont les équipes PARADIS et SCALAPLIX du Labri, l'institut européen de chimie et biologie (IECB), le laboratoire de structures et réactivité des systèmes moléculaires complexes (SRSMC) et la station Méditérannée de l'environnement littoral (CNRS/SMEL).

L'objectif du projet EPSN est de concevoir une plate-forme permettant de piloter par la visualisa-

tion des applications numériques distribuées (*computational steering*). Le but est de pouvoir interagir avec les calculs pendant l'exécution, de façon à diriger l'application en fonction des résultats intermédiaires obtenus. Cette approche est particulièrement intéressante pour des applications complexes qui peuvent tirer profit d'un centre de réalité virtuelle. Le projet EPSN mène au développement de l'environnement *Epsilon* qui permet de réaliser la visualisation et l'interaction avec les applications en restant très peu intrusif. Les communications pour le pilotage et la visualisation sont assurées par CORBA.

L'utilisation de PadicoTM dans EPSN présente de multiples atouts. Tout d'abord, PadicoTM offre une implémentation CORBA haute performance utilisable sur différents types de réseaux, et peut emprunter les réseaux haute performance quand ils sont disponibles. L'utilisation de CORBA dans PadicoTM sur les réseaux haute performance apporte à la fois une faible la latence et un haut débit, propriétés profitables pour les interactions en temps réel. En effet, l'interaction en temps réel avec un utilisateur nécessite des communications avec une latence extrêmement faible pour que l'interaction soit confortable pour l'utilisateur. De plus, le volume de données transmis pour la visualisation pouvant être important à un rythme soutenu, il est intéressant de pouvoir exploiter efficacement les réseaux haut débit.

Ensuite, les codes applicatifs numériques peuvent être parallèles, par exemple basés sur MPI. Le problème de l'utilisation conjointe de MPI et CORBA sur le même réseau dans les mêmes processus est un autre problème résolu par PadicoTM.

PadicoTM permet donc au projet EPSN d'une part d'élargir ses ressources utilisables vers des machines parallèles et grappes, et d'autre part d'élargir les applications numériques prises en comptes vers les codes parallèles. L'avantage apporté par l'utilisation de PadicoTM est donc appréciable.

11.5 L'environnement DIET et le projet GASP

DIET (*Distributed Interactive Engineering Toolbox*) est un prototype logiciel développé initialement dans le projet REMAP du LIP et l'équipe "Distribution et Parallélisme" du LIFC. Le projet GASP (*Grid Application Service Provider*) est un projet du Réseau National des Technologies Logicielles du ministère de l'industrie. L'objectif du projet GASP est le développement de DIET et ses applications. Les partenaires de GASP sont le projet REMAP du LIP, le projet ALGORILLE du LORIA, l'équipe DP du LIFC, le LST et *Sun Microsystems*.

DIET est un ASP (*Application Service Provider*). Il s'agit d'une approche client-serveur des grilles de calcul. L'architecture générale est constituée de cinq entités : des clients soumettent des problèmes numériques ; des serveurs sont à l'écoute de requêtes des clients et disposent de bibliothèques spécifiques (BLAS, LAPACK, ScaLAPACK, PETSc) pour les calculs ; une base de données mémorise les ressources logicielles et matérielles à disposition; un ordonnanceur choisit quelles ressources attribuer à chaque requête ; un superviseur observe l'état courant des ressources.

Cette approche de la programmation des grilles de calcul demande l'utilisation d'un exécutif pour l'échange des requêtes (CORBA, GRID RPC, etc.) qui peut avoir à cohabiter avec un autre exécutif lorsque les codes de calculs numériques sont parallèles (par exemple MPI avec ScaLAPACK). Dans ce cas, PadicoTM offre plusieurs exécutifs simultanément dans le même processus sur les mêmes ressources. De plus, l'utilisation de PadicoTM permet aux messages de requêtes — potentiellement volumineux, car ils transportent les données du problème — d'utiliser les réseaux rapides des grappes de calculateurs. La version sur PadicoTM est destinée à devenir la version principale de DIET.

11.6 Projet ALTA

Le projet ALTA [21] est une action conjointe ARC-ACI — Action de Recherche Coopérative INRIA et Action Concertée Incitative GRID 2002 du ministère de la recherche. Le projet ALTA est intitulé "*Algorithmes itératifs asynchrones et protocoles de communication à tolérance de perte ajustable*". Les partenaires impliqués sont le projet PARIS de l'IRISA, le projet RUNTIME du Labri, et l'équipe DP du LIFC.

Le but du projet ALTA est d'étudier l'utilisation de protocoles à tolérance de perte ajustable, tels que VRP [14], pour des applications de calcul numérique basées sur des algorithmes itératifs asynchrones.

L'origine du projet ALTA est le constat que la fiabilisation des réseaux coûte cher. Quand des paquets sont perdus par le réseau, les algorithmes de fiabilisation tels que TCP ont un impact très négatif sur le débit. Dans ces conditions, il est clair que l'utilisation d'un protocole tolérant un certain degré de pertes permettrait d'augmenter le débit utile de manière significative. Toutefois, l'utilisation d'un protocole à tolérance de perte n'est possible que si l'application tolère les pertes.

Le premier objectif du projet ALTA est de faire évoluer l'implémentation préliminaire de VRP dans PadicoTM. Il s'agit avant tout de définir une interface de programmation de niveau applicatif qui permette un contrôle des paramètres du protocole et qui soit adaptée aux schémas d'utilisation par les applications.

Le deuxième objectif est d'appliquer la tolérance de pertes aux algorithmes itératifs asynchrones. Les algorithmes itératifs réalisent leurs calculs par itérations successives pour converger vers le résultat. Les algorithmes itératifs asynchrones ont la particularité que plusieurs nœuds impliqués dans le calcul en parallèle peuvent effectuer les itérations sans se synchroniser ; ils échangent des messages pour propager leurs résultats sans que les différents nœuds effectuent la même itération au même moment. La tolérance de pertes pour les communications entre nœuds d'un algorithme itératif asynchrone se traduit par des mises à jour partielles : les zones perdues des messages ne sont pas mises à jour sur le récepteur, et gardent la valeur de l'itération précédente ; ce n'est pas gênant du fait que ces algorithmes tolèrent l'asynchronisme, et ceci n'empêche pas la convergence.

Il est très intéressant d'implémenter VRP dans PadicoTM. Dans le cas de réseaux à fort taux de perte, le protocole VRP est mis en œuvre. Dans le cas où la même application est déployée sur un autre réseau, comme par exemple un réseau haute performance sur une grappe, il est possible le cas échéant d'offrir l'interface de VRP au-dessus de méthodes de communications fiables. Ceci permet de ne pas avoir à modifier l'application lors du passage d'un réseau à un autre, et de profiter de toutes les méthodes de communication de PadicoTM.

11.7 Conclusion

Nous avons présenté dans ce chapitre des utilisations de PadicoTM dans des situations diverses. Nous avons vu plusieurs exemples : PadicoTM apporte une solution à l'utilisation simultanée de CORBA + MPI sur tout type de réseau pour PaCO++ et GRIDCCM ; il permet des combinaisons d'exécutifs variées dans GRID-RMI ; il autorise l'utilisation d'un exécutif différent pour le contrôle et les interactions d'une part, et le parallélisme des calculs d'autre part, dans EPSN et DIET ; et enfin il permet de façon flexible d'avoir à disposition tout un éventail de méthodes de communications utilisables de façon interchangeables dans ALTA.

Les exécutifs du répartis étant de plus en plus utilisés pour la programmation des grilles de calcul — pour la visualisation, le contrôle, ou le couplage de codes —, leur utilisation sur des réseaux haute performance, éventuellement de façon conjointe avec des exécutifs du parallélisme, les problèmes résolus par la plate-forme PadicoTM sont de plus en plus fréquemment rencontrés au cours du développement d'environnements pour les grilles.

Chapitre 12

Conclusion et perspectives

Conclusion

Contexte d'étude. Les grilles de calcul sont un domaine émergent, qui regroupe des personnes venant de tous horizons, avec des visions parfois radicalement différentes de ce qu'est une grille et de la façon de les programmer. En effet, les grilles de calcul sont un point de convergence entre le monde des systèmes répartis et du parallélisme, entre la communauté des réseaux et celle du calcul haute performance. Cette ambivalence est l'essence-même des grilles, ce qui les distingue des systèmes purement répartis ou purement parallèles.

D'un point de vue architectural, une grille est constituée de ressources de calcul variées — supercalculateurs, grappes, stations de travail — régies généralement selon les principes du parallélisme. L'assemblage de ces diverses machines est hétérogène, distribué à grande échelle sur plusieurs sites, et dynamique, qui sont des traits caractéristiques des systèmes répartis. Nous retrouvons le carrefour entre les systèmes répartis et le parallélisme dans les modèles de programmations. Selon les cas, une grille est vue et programmée comme un système parallèle ou comme un système réparti. Des modèles de programmation récents proposent une combinaison de méthodes des systèmes répartis et parallèles.

Devant la diversité des ressources et la diversité des approches, il ne semble pas approprié de chercher à imposer un modèle de programmation particulier ou des ressources spécifiques. Un environnement de programmation des grilles se doit de supporter les différents types de ressources, les différents modèles de programmation existants, et de laisser le choix du modèle de programmation — réparti, parallèle ou une combinaison — indépendamment du type de ressource réellement utilisées à l'exécution.

Les communications représentent un enjeu majeur dans l'utilisation des grilles. Elles sont un goulot d'étranglement pour les performances des applications ; la connectivité est également un aspect problématique, l'interconnexion n'allant pas de soi entre les réseaux du réparti et du parallélisme conçus dans des optiques différentes. Bien souvent, la solution proposée pour les communications sur les grilles est centrée sur MPI et uniquement les modèles de programmation parallèles, ou sur les *Web Services* et uniquement les modèles de programmation du réparti. Nous pensons que ces approches des communications sont trop pauvres et ne permettent pas d'exprimer la richesse des besoins des applications potentielles des grilles.

Contribution. Nous avons proposé dans ce document une architecture pour plate-forme de communication qui prenne en compte les caractéristiques à la fois de l'architecture des grilles et des applications. De part la diversité des ressources et des exécutifs, notre proposition de plate-forme de communication agit comme un véritable centre de communication, permettant aux exécutifs d'exploiter les ressources disponibles variées. Cette plate-forme permet d'utiliser n'importe quel exécutif sur

n'importe quel réseau supporté ; la plate-forme se charge de réaliser l'adaptation entre l'abstraction présentée par l'interface offerte par le réseau et l'interface utilisée par l'exécutif. Ceci découple le paradigme de l'exécutif du paradigme du réseau : le choix de l'exécutif est dicté par le concepteur de l'application, le choix de la méthode utilisée à bas niveau est dicté par la nature des ressources indépendamment de l'exécutif choisi pour l'application.

Les caractéristiques principales de l'architecture de plate-forme de communication que nous avons proposée dans ce document sont :

– l'accès arbitré aux ressources, pour permettre l'utilisation simultanée de plusieurs exécutifs ;
– l'adaptation d'abstraction, pour découpler le paradigme offert au niveau applicatif du paradigme proposé par les ressources ;
– la virtualisation des interfaces de communication, pour intégrer le code existant — exécutifs et applications — sans le modifier en lui offrant, au-dessus d'un éventail de ressources élargi, l'interface dont il est familier.

À ces trois aspects liés aux communications s'ajoute l'aspect transversal de la flexibilité apporté par l'utilisation de modules assemblables qui permettent une variété de méthodes de communications avec un choix effectué à l'exécution, et le paramétrage par l'utilisateur des stratégies d'assemblage. Les points forts de ce modèle de plate-forme de communication sont notamment :

support d'exécutifs variés — Étant donnée la profusion des exécutifs existants et susceptibles d'être utilisés par des applications exécutées sur les grilles, il nous a semblé nécessaire de supporter une large palette d'exécutifs à offrir aux concepteurs d'applications : MPI et SOAP, mais aussi CORBA, RMI Java, MVP, ICE, HLA et éventuellement d'autres. L'interface de ces exécutifs n'étant pas modifiée, le portage d'une application sur notre plate-forme est transparent.

support de plusieurs exécutifs simultanément — Certains modèles de programmation pour les grilles demandent l'utilisation simultanée de plusieurs exécutifs, typiquement un exécutif parallèle et un exécutif réparti. L'utilisation simultanée de plusieurs exécutifs est susceptible de causer des conflits d'accès aux ressources et de ne pas fonctionner dans certains cas. Le modèle de plate-forme que nous proposons prend en compte ce besoin dès sa conception.

support de toute la hiérarchie de réseaux — Une caractéristique essentielle de l'architecture des grilles est la présence de différentes classes de réseaux (SAN, LAN, WAN) avec des services offerts, des performances, et des méthodes d'utilisation particulièrement variés. L'architecture que nous proposons est conçue pour être portable sur un large éventail de technologies réseaux et pour exploiter chaque type de réseau au mieux.

méthodes de communication extensibles — Pour une évolutivité par rapport aux différents types de réseaux, nous avons privilégié une approche modulaire qui permet d'ajouter facilement de nouvelles méthodes de communication et supporter de nouveaux types de réseaux.

plate-forme multi-paradigme — Notre approche de l'adaptation de paradigme permet d'utiliser chaque réseau avec diverses interfaces abstraites obéissant chacune à un paradigme — répartie, parallèle à mémoire distribuée, ou parallèle à mémoire partagée. Les compromis nécessaires pour passer d'un paradigme à l'autre ne sont réalisés que lorsqu'un utilisateur demande un exécutif selon un paradigme différent de celui de la ressource. Aucun paradigme particulier n'est globalement imposé ni privilégié.

adéquation aux contraintes de chaque paradigme — L'adaptation des paradigmes est réalisée dans le respect des contraintes et priorités de chaque paradigme. Ainsi, par exemple pour les exécutifs du réparti, nous apportons un soin tout particulier à l'interopérabilité avec les exécutifs standard qui n'utilisent pas notre plate-forme.

Mise en œuvre. Nous avons mis en œuvre notre architecture de plate-forme de communication dans PadicoTM, plate-forme de communication pour les grilles. PadicoTM est organisé selon le modèle proposé, à savoir un niveau d'arbitrage, un niveau d'adaptation d'abstraction, et un niveau de virtualisation d'interface. Nous avons porté des exécutifs variés dans PadicoTM : une implémentation MPI, plusieurs implémentations CORBA, une implémentation SOAP, une machine virtuelle Java, une MVP,

une implémentation HLA. Ces exécutifs ont à leur disposition par PadicoTM les réseaux accessibles par TCP/IP et par Madeleine ; l'interopérabilité par TCP/IP est assurée de façon transparente vers les exécutifs sans PadicoTM. De plus, des méthodes de communication additionnelles pour des cas spécifiques sont à leur disposition, comme les *ParallelStreams* et la compression AdOC.

Nous avons évalué les performances de PadicoTM et de divers exécutifs dans PadicoTM. Les résultats obtenus sont probants et montrent qu'il est possible de réaliser une implémentation efficace du modèle que nous avons proposé.

La plate-forme PadicoTM est utilisée dans divers projets, tels que PaCO++ et GRIDCCM, EPSN, HydroGRID et DIET. Elle a de plus fait l'objet d'un dépôt à l'Agence pour la Protection des Programmes et est disponible au téléchargement.

Perspectives

Ces travaux débouchent sur des prolongements possibles. À court terme, il s'agit principalement d'élargir la palette de méthodes de communication et d'étudier de façon plus approfondie les contentions. À moyen terme, il serait intéressant d'étudier les stratégies d'adaptation et de déploiement.

Méthodes de communication. Tout d'abord, les infrastructures de sécurité, bien que prévues dans le modèle, n'ont pas été implémentées. Au-delà de la seule implémentation, la sécurité apporte de nouvelles problématiques. Nous avons proposé de positionner le chiffrement et l'authentification dans des modules qui présentent une interface abstraite de type réparti. Cependant, ceci ne résout pas la problématique de la gestion des clés et certificats ni la délégation d'autorité. Les travaux autour de la sécurité menés par l'Alliance Globus étant adoptés en tant que standards, une prise en compte de la sécurité dans le contexte des grilles passe sans doute par une adoption de GSI, l'infrastructure de sécurité de Globus.

Les travaux autour de la tolérance de perte ajustable et du protocole VRP méritent également un prolongement au-delà de l'implémentation embryonnaire mise en œuvre dans PadicoTM et de ses résultats préliminaires prometteurs. Il semble nécessaire de mener d'autres expérimentations pour peaufiner la modélisation du comportement du réseau et améliorer ainsi le protocole. Des efforts doivent également être consentis au niveau de son interface de programmation de façon à mieux correspondre aux applications qui tolèrent les pertes. Dans le cadre des algorithmes itératifs asynchrones, ces travaux font l'objet de l'action conjointe ARC-ACI "ALTA" [21] en partenariat avec le projet RUNTIME du Labri et le projet "Distribution et parallélisme" du LIFC.

Étude de la concurrence, des contentions et optimisations. À un niveau plus proche de l'implémentation, même si les performances obtenues par PadicoTM sont convaincantes, une grande partie du code est à l'état de prototype de recherche et n'est pas optimisé. En particulier l'adaptateur *VLink/MadIO*, offrant une interface de type réparti sur les réseaux haute performance, présente une latence qui pourrait probablement être améliorée en optimisant le code ; le débit pourrait probablement être également amélioré, le décrochement lors du passage en mode "rendez-vous" étant sensible sur cette implémentation, alors qu'il peut être rendu imperceptible d'après des observations faites sur d'autres plates-formes.

La concurrence pourrait faire l'objet d'une étude plus approfondie. Nous avons limité notre étude de la concurrence au cas classique, dans lequel la scrutation sur plusieurs réseaux et par plusieurs exécutifs a lieu simultanément, mais nous avons étudié seulement marginalement le cas où plusieurs communications ont effectivement lieu en même temps. La contention devrait être étudié de façon plus approfondie en examinant de façon précise l'influence des communications les unes sur les autres, et pas seulement des scrutations.

Adaptabilité. Nous voyons également des perspectives à moyen terme. Tout d'abord, notre architecture pour plate-forme de communication offre la flexibilité et le paramétrage à l'utilisateur. Cependant, les algorithmes qui décident des protocoles, réseaux et méthodes de communication sont

rudimentaires. Ils sont suffisants pour les cas que nous avons rencontrés en pratique mais peuvent ne pas passer à l'échelle pour des systèmes plus complexes. De plus, ces algorithmes sont basés sur un système de préférences données par l'utilisateur. La plate-forme ne fournit que les mécanismes pour changer les paramètres, pas les algorithmes de décision. Pourtant, la plupart du temps ces données pourraient être obtenues de façon automatique par observation de l'état du réseau, de la charge des machines, ou du schéma de communication constaté pour l'application en cours. C'est le domaine d'application des *systèmes adaptables* qui a pour but de prendre automatiquement des décisions d'adaptation de l'exécution à l'environnement observé. Il serait intéressant d'étudier l'application des méthodes de décision issues des systèmes adaptables à notre plate-forme de communication.

Déploiement. Le déploiement et la gestion de ressources sont d'autres enjeux de taille. L'architecture que nous avons proposé prend en charge les communications lors de l'exécution. L'ensemble des machines et réseaux à utiliser sont une donnée de départ. Le déploiement considère le problème en amont. Il s'agit de disposer d'un annuaire de ressources avec leurs caractéristiques, de maintenir à jour une base de donnée de leur état de disponibilité, et d'offrir des mécanismes d'allocation de ressource, c'est-à-dire de décision des ressources à attribuer à une application. La décision peut être prise en fonction de nombreux paramètres, en particulier l'état courant des ressources et leur adéquation aux besoins de l'application à lancer, mais peut aussi prendre en compte des paramètres comme la disponibilité de code déjà compilé pour certaines architectures. Le placement des nœuds de l'application sur les machines est effectué en fonction du schéma de communication de l'application, des performances des réseaux, et de contraintes d'équilibrage de charge. Ensuite a lieu le déploiement proprement dit, avec éventuellement le téléchargement du code et de ses dépendances, des données, et enfin le lancement des processus. Il nous semble tout-à-fait pertinent d'étudier les interactions entre les mécanismes de déploiement et la plate-forme de communication.

Ouverture vers le pair-à-pair. Ces travaux ouvrent la voie à d'autres directions de recherche. La première direction qu'il nous semble intéressant d'explorer est le domaine des réseaux pair-à-pair. Il est envisageable d'utiliser une approche de type pair-à-pair pour maintenir des annuaires de ressources qui tiennent compte de la volatilité des ressources. En effet, un des problèmes principaux des grilles est une gestion de ressources qui tiennent compte à la fois de la grande volatilité des nœuds, mais en même temps d'un passage à grande échelle. L'approche pair-à-pair permet de conserver la notion de "monde connu" telle que présente dans notre modèle de plate-forme de communication, tout en s'affranchissant de la faiblesse qu'implique la connaissance d'un état global ou d'un annuaire centralisé.

L'engouement pour le pair-à-pair sur les grilles de calcul ne se confine cependant pas à la seule maintenance d'un annuaire de ressources. Des projets existent pour utiliser des réseaux pair-à-pair pour stocker des données à grande échelle de façon distribuée, et pour réaliser des calculs. Toutes ces utilisations du pair-à-pair pour les grilles ont, au même titre que les autres utilisations plus "conventionnelles", des exigences vis-à-vis des communications. Il nous paraît donc tout-à-fait adéquat d'étudier dans quel mesure l'architecture de plate-forme de communication pour les grilles que nous avons proposée peut être utilisée, étendue ou adaptée, pour ces utilisations futures des grilles.

Généralisation. Enfin, il serait sans doute intéressant d'étudier dans quelle mesure notre approche est généralisable. Plusieurs pistes sont à étudier. Tout d'abord, nous avons restreint notre étude au cas des grilles *de calcul*, en nous focalisant plus particulièrement sur les applications de simulation numérique. Or, l'utilisation des grilles s'élargit à de nombreux autres domaines ; les bases de données passent à grande échelle, ou encore les applications de création vidéo permettant maintenant de collaborer en temps réel sur *internet* et d'agréger la puissance de plusieurs PC sur un réseau local pour des calculs parallélisés. Il serait probablement pertinent d'étudier dans quelle mesure notre approche peut prendre en compte ces domaines qui combinent aussi des aspects du calcul réparti et du calcul parallèle.

Il est également légitime de se poser la question d'intégrer la plate-forme de communication au système. Les pistes envisageables sont l'intégration en tant que bibliothèque fournie avec le système,

ou même directement une intégration dans le noyau d'un système d'exploitation distribué. En effet, en première approche, il semble que les services de communication offerts par un système d'exploitation distribué aux applications, et les communications utilisées par le système lui-même, peuvent tout-à-fait être organisées selon le modèle que nous proposons. Il nous semble donc intéressant d'étudier une intégration de cette organisation des communications dans un système d'exploitation.

Bibliographie

Les premières références bibliographiques correspondent à nos propres publications. Elles sont listées à la page 3.

[21] ALTA: Asynchronous loss tolerant algorithms. `http://www.irisa.fr/alta/`.

[22] Cheetah: A library for remote method invocation layered on a message passing interface. `http://www.acl.lanl.gov/cheetah/`.

[23] Folding@home, distributed computing. `http://folding.stanford.edu/`.

[24] The gnome project. `http://www.gnome.org/`.

[25] MICO, an OpenSource CORBA implementation. `http://www.mico.org/`.

[26] openSSL: the open source toolkit for TLS/SSL. `http://www.openssl.org/`.

[27] ORBacus(tm) for C++ and Java. World Wide Web document, `http://www.orbacus.com/products/orbacus.html`.

[28] PAWS: Parallel application workspace. `http://www.acl.lanl.gov/paws/`.

[29] PM2 High Perf. World Wide Web document, `http://www.pm2.org/`.

[30] SETI@home: The search for extraterrestrial intellignece. `http://setiathome.berkeley.edu/`.

[31] Sun ONE Grid Engine. `http://www.sun.com/gridware/`.

[32] The Globus Alliance. `http://www.globus.org/`.

[33] The Object Management Group. World Wide Web document, `http://www.omg.org`.

[34] TOP500 supercomputers sites. `http://www.top500.org/`.

[35] Unicore forum. `http://www.unicore.org/`.

[36] Portable operating system interfaces (posix(^{r})) - part 1: System application program interface (api) - amendment 1: Realtime extension [c language]. IEEE Standard number P1003.1b-1993 (POSIX.1b), 1993. Anciennement P1003.4.

[37] Open Systems Interconnection—basic reference model. ISO/IEC 7498, ITU-T Rec. X.200, 1994.

[38] Protocol independent interfaces for process-to-process communication. IEEE Standard number P1003.1g (POSIX.1g), Janvier 2000. Identique à ISO/IEC 9945-1:1996.

[39] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing Journal*, 28(5):749–771, Mai 2002.

[40] 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, Février 1996.

[41] ANSI/VITA. Myrinet-on-vme protocol specification. Standard no 26-1998, 1998.

[42] Gabriel Antoniu and Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, volume 2026 of *Lect. Notes in Comp. Science*, pages 55–70, San Francisco, Avril 2001. Held in conjunction with IPDPS 2001. IEEE TCPP, Springer-Verlag.

[43] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, 1999.

[44] InfiniBand Trade Association. Infiniband architecture specification release 1.1, 2002.

[45] OmniORB Home Page. AT&T Laboratories Cambridge, `http://www.omniorb.org`.

[46] O. Aumage, G. Mercier, and R. Namyst. MPICH/Madeleine: a true multi-protocol MPI for high-performance networks. In *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, page 51, San Francisco, Avril 2001. IEEE.

[47] Olivier Aumage. *Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes*. PhD thesis, École Normale Supérieure de Lyon, Spetembre 2002.

[48] Olivier Aumage, Luc Bougé, Lionel Eyraud, and Raymond Namyst. Communications efficaces au sein d'une interconnexion hétérogène de grappes : Exemple de mise en oeuvre dans la bibliothèque Madeleine. Soumis pour publication à *Calculateurs parallèles*. Numéro spécial Métacomputing: calcul réparti à grande échelle, Mars 2001.

[49] Thomas Beisel, Edgar Gabriel, and Michael Resch. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, chapter An Extension to MPI for Distributed Computing on MPPs, pages 75–83. Lecture Notes in Computer Science. Springer, 1997.

[50] Robert D. Blumofe, Christopher F. Joerg, Bradley Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *In 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP '95*, pages 207–216, Santa Barbara, Californie, Juillet 1995.

[51] OpenMP Architecture Review Board. OpenMP C and C++ application program interface version 2.0, Mars 2002.

[52] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. S., and W.-K. Su. Myrinet : A gigabit-per-second local area network. *IEEE-Micro*, 15(1):29–36, Février 1995.

[53] L. Bougé, J.-F. Méhaut, and R. Namyst. Efficient communications in multithreaded runtime systems. In *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 468–482, San Juan, Puerto Rico, Avril 1999. In conj. with IPPS/SPDP 1999. IEEE TCPP and ACM SIGARCH, Springer-Verlag.

[54] R. Braden. Requirements for internet hosts – communication layers. Technical report, Octobre 1989.

[55] John Bresnahan. The extensible input output library for the globus toolkit (tm). `http://www-unix.mcs.anl.gov/~bresnaha/`.

[56] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *LNCS*, pages 907–910, Paderborn, Germany, Août 2002. Springer-Verlag.

[57] Gerson Cavalheiro, François Galilée, and Jean-Louis Roch. Athapascan-1: Parallel programming with asynchronous tasks. In *Yale Multithreaded Programming Workshop*, Yale, USA, 1998.

[58] Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. Number 2246. O'Reilly, Février 2002.

[59] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[60] Steve Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw. Resource management in legion. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'99), in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS'99)*, APR 1999.

[61] J. Crowcroft, I. Wakeman, and Z. Wang. Layering considered harmful. *IEEE Network*, 6(1), Janvier 1992.

[62] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *Proc. 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC10)*. IEEE Press, 2001.

[63] Karl Czajkowski, Ian Foster, Nicholas Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing,*, pages 62–82, 1998.

[64] V. Danjean, R. Namyst, and R. Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, volume 1800 of *Lect. Notes in Comp. Science*, pages 1160–1167, Cancun, Mexico, Mai 2000. In conjunction with *IPDPS 2000. IEEE TCPP and ACM*, Springer-Verlag.

[65] Vincent Danjean. Réactivité aux événements d'entrées/sorties dans les environnements multithreads. In *Actes des 14èmes Rencontres francophones du parallélisme (RenPar 14)*, Hammamet, Tunisie, Avril 2002.

[66] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, Avril 1988.

[67] Thomas DeFanti, Ian Foster, Michael Papka, Rick Stevens, and Tim Kuhfuss. Overview of the i-way: Wide area visual supercomputing. *International Journal of Supercomputer Applications*, 10(2):123–130, 1996.

[68] T. Dierks and C. Allen. The TLS protocol version 1.0. Ietf request for comment 2246, Janvier 1999.

[69] Nhu-Tung Doan. Optimizing inter-ORB communication for a high-performance distributed object broker. Master's thesis, IRISA, Octobre 2000.

[70] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne-Marie Meritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–75, Mars 1998.

[71] Dan Egnor. `liboop` home page. `http://www.iboop.org/`.

[72] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Ietf request for comment 2068, Janvier 1997.

[73] Jeffrey M. Fischer and Milos D. Ercegovac. A component framework for communication in distributed applications. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 647–654, Cancun, Mexique, Mai 2000. IEEE Computer Society.

[74] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*, pages 365–375, 1997.

[75] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[76] International Organization for Standardization. Fibre distributed data interface (FDDI). Standard ISO 9314-1:1989.

[77] Frame Relay Forum. Pvc user-to-network interface (UNI) implementation agreement. Implementation Agreement number FRF.1.2, Juillet 2000.

[78] High Performance Fortran Forum. High performance fortran version 2.0, Janvier 1997.

[79] Message Passing Interface Forum. Message passing interface standard. Technical report, University of Tennessee, Mai 1994.

[80] The ATM Forum. ATM user-network interwork interface (UNI) specification version 4.1. Standard af-arch-0193.000, Novembre 2002.

[81] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, , and S. Tuecke. Wide-area implementation of the message passing interface. *Parallel Computing*, 24(12):1735–1749, 1998.

[82] I. Foster, J. Geisler, and S. Tuecke. MPI on the I-WAY: A wide-area, multimethod implementation of the message passing interface. In *MPI Developers Conference*, pages 10–17, 1996.

[83] I. Foster, C. Kesselman, J. Nick, and S. 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, Juin 2002.

[84] Ian Foster. What is the grid? a three point checklist. *GRIDtoday*, 1(6), Juillet 2002. `http://www.gridtoday.com/02/0722/100136.html`.

[85] Ian Foster, Jonathan Geisler, Carl Kesselman, and Steven Tuecke. Managing multiple communication methods in high-performance networked computing systems. *J. Parallel and Distributed Computing*, 40:35–48, 1997.

[86] Ian Foster, Jonathan Geisler, Bill Nickless, Warren Smith, and Steven Tuecke. Software infrastructure for the i-way high performance distributed computing experiment. In *Proc. 5th IEEE Symposium on High Performance Distributed Computing*, pages 562–571, 1997.

[87] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[88] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[89] Ian Foster, Carl Kesselman, and Steven Tuecke. Nexus: Runtime support for task-parallel programming languages. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[90] Ian Foster, Carl Kesselman, and Steven Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[91] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Intl Journal of Supercomputer Applications*, 15(3), 2001.

[92] Ian T. Foster and K. Mani Chandy. FORTRAN M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[93] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Press, 1994.

[94] Wolfgang Gentzsch. Response to ian foster's "what is the grid?" [84]. *GRIDtoday*, 1(8), Août 2002. `http://www.gridtoday.com/02/0805/100191.html`.

[95] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B. D. Jonhsen, H. Kohmann, R. Nordstrom, and P. Werner. Low-level SCI software functional specification: Software infrastructure for SCI (SISCI). Technical Report Deliverable D.1.1.1, Esprit Project 23174, MAR 1999.

[96] Vincent Néri Gilles Fedak, Cécile Germain and Franck Cappello. XtremWeb: A generic global computing system. In *CCGRID2001, workshop on Global Computing on Personal Devices*. IEEE Press, Mai 2001.

[97] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *THE JAVA LANGUAGE SPECIFICATION, Second Edition*. Addison-Wesley, Juin 2000. Disponible au téléchargement : `http://java.sun.com/docs/books/jls/`.

[98] Andrew Grimshaw, Adam Ferrari, and Emily West. *Parallel Programming Using C++*, chapter Mentat, pages 383–427. The MIT Press, Cambridge, Massachusetts, 1996.

[99] Andrew Grimshaw, Wm Wulf, and the whole Legion team. Legion: The next logical step toward the world-wide virtual computer. *Communications of the ACM*, Janvier 1997.

[100] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Spetembre 1996.

[101] William Gropp and Ewing Lusk. The second-generation ADI for the MPICH implementation of MPI. MPICH working note, ANL/MCS, 1996.

[102] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[103] William D. Gropp and Barry Smith. Chameleon parallel programming tools user manual. Technical Report ANL-93/23, Argonne National Laboratory, Argonne, Illinois, 1993.

[104] The Open Group. Networking services, issue 4. Technical Standard C438, Spetembre 1994.

[105] The Open Group. DCE 1.1: Remote procedure call. Technical Standard C706, Août 1997.

[106] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[107] Haifa MD Development Lab IBM Corporation. Infiniband access application programming interface specification version 1.2, Avril 2002.

[108] IEEE. Standard for Scalable Coherent Interface (SCI). Standard no. 1596, Août 1993.

[109] IEEE. Ieee standard for modeling and simulation (m&s) high level architecture (hla) - framework and rules. IEEE Standard number 1516, 2000.

[110] Joel D. Daniels II. Improving network performance in computational grid applications. Bachelor of science honors thesis, Univeristy of NewHampshire, Octobre 2002.

[111] Yuji Imai, Toshiaki Saeki, Tooru Ishizaki, and Mitsushiro Kishimoto. CrispORB: High performance CORBA for system area network. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 11–18, 1999.

[112] INRIA. Inria – gelato consortium. `http://www.inrialpes.fr/gelato/`.

[113] International Telecommunications Union (ITU). Abstract syntax notation one (asn.1) specification of basic notation. ITU-T Rec. X.680, ISO/IEC 8824-1:2002, 2002.

[114] Emmanuel Jeannot, Bjorn Knutsson, and Mats Bjorkmann. Adaptive online data compression. In *IEEE High Performance Distributed Computing (HPDC'11)*, Edinburgh, Scotland, Juillet 2002.

[115] Elizabeth Johnson and Dennis Gannon. HPC++: Experiments with the parallel standard template library. In *International Conference on Supercomputing*, pages 124–131, 1997.

[116] Nicholas T. Karonis, Bronis R. de Supinski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 377–386, Cancun, Mexique, Mai 2000.

[117] Dan Kegel. The c10k problem. `http://www.kegel.com/c10k/`,2003.

[118] S. Kent and R. Atkinson. Security architecture for the internet protocol. Ietf request for comment 2401, Novembre 1998.

[119] Thilo Kielmann, Henri E. Bal, Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Rutger Hofman, Ceriel Jacobs, and Kees Verstoep. The albatross project: Parallel application support for computational grids. In *1st European GRID Forum Workshop*, pages 341–348, Poznan, Poland, Avril 2000.

[120] Thilo Kielmann, Rutger F.H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A.F. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, Atlanta, GA, Mai 1999.

[121] Bobby Krupczak, Mostafa Ammar, and Ken Calvert. Multi-subsystem protocol architectures: Motivation and experience with an adapter-based approach. In *Proceedings of the IEEE INFOCOM*, 1996.

[122] Fred Kuhns, Douglas Schmidt, and David Levine. The design and performance of a real-time I/O subsystem. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS99)*, Vancouver, Canada, Juin 1999.

[123] Ch. Kurmann and T. Stricker. Zero-copy for corba – efficient communication for distributed object middleware. In *Proceedings of the 12th International Symposium on High Performance Distributed Computing (HPDC12)*, Seattle, Washington, Juin 2003.

[124] Dawid Kurzyniec and Vaidy Sunderam. Efficient cooperation between java and native codes – jni performance benchmark. In *2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, Juin 2001.

[125] Dawid Kurzyniec, Vaidy Sunderam, and Mauro Migliardi. On the viability of component frameworks for high performance distributed computing: A case study. In *IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, Edimburg, Scotland, Juillet 2002.

[126] J. Lemon. Kqueue: A generic and scalable event notification facility. FreeBSD Project.

[127] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Novembre 1989.

[128] Davide Libenzi. "`/dev/epoll`" home page. `http://www.xmailserver.org/linux-patches/nio-improve.html`.

[129] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[130] Sai-Lai Lo and Steve Pope. The implementation of a high performance ORB over multiple network transports. rapport de recherche, Olivetti & Oracle Laboratory, Cambridge, Mars 1998.

[131] Steven S. Lumetta, Alan M. Mainwaring, and Devis E. Culler. Multi-protocol active messages on a cluster of smp's. In *Supercomputing (SC'97)*, 1997.

[132] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel J. H. Jacobs, and Rutger F. H. Hofman. Efficient java RMI for parallel programming. *Programming Languages and Systems*, 23(6):747–775, 2001.

[133] Alan Mainwaring and David Culler. Active message applications programming interface and communication subsystem organization. Technical report, University of California at Berkeley, 1996.

[134] Mauro Migliardi and Vaidy Sunderam. The harness metacomputing framework. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, USA, Mars 1999.

[135] Mauro Migliardi and Vaidy Sunderam. Heterogeneous distributed virtual machines in the harness metacomputing framework. In *Proceedings of the Heterogeneous Computing Workshop of IPPS/SPDP 1999*, pages 60–73, S. Juan de Puerto Rico, Avril 1999. IEEE Computer Society Press.

[136] Myricom. GM: A message-passing system for myrinet networks. `http://www.myri.com/scs/`, Juillet 2003.

[137] Myricom. Myrinet Express (MX): A high performance, low-level, message-passing interface for myrinet. `http://www.myri.com/scs/`, Juillet 2003. Pre-release 25 juillet 2003.

[138] Raymond Namyst and Jean-François Méhaut. *Marcel : Une bibliothèque de processus légers*. LIFL, Univ. Sciences et Techn. Lille, 1995.

[139] Raymond Namyst and Jean-François Méhaut. PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, Spetembre 1995.

[140] Anand Natrajan, Marty Humphrey, and Andrew Grimshaw. Grids: Harnessing geographically separated resources in a multi-organisational context. Presented at High Performance Computing Systems 2001, 2001.

[141] Object Management Group. Request For Proposal: Data Parallel Application Support for CORBA, Mars 2000.

[142] Institute of Electrical and Electronics Engineers. CSMA/CD access method, 2002. Standard IEEE 802.3.

[143] Institute of Electrical and Electronics Engineers. Token ring access method and physical layer specifications, 2002. Standard IEEE 802.5.

[144] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems (TOCS)*, 10(2):110–143, Mai 1992.

[145] OMG. Benchmark PSIG, White Paper on Benchmarking. OMG Document bench/99-12-01, Décembre 1999.

[146] OMG. The Common Object Request Broker: Architecture and Specification (Revision 2.5). OMG Document formal/01-09-34, Spetembre 2001.

[147] Onera. http://www.cert.fr/CERTI/.

[148] Scott Parkin, Vijay Karamcheti, and Andrew A. Chien. Fast messages: Efficient, portable, communication for workstations clusters and mpps. *IEEE Concurrency*, 5(2):60–73, Avril 1997.

[149] Steve Pope and Sai-Lai Lo. The implementation of a native ATM transport for a high performance ORB. rapport de recherche, Olivetti & Oracle Laboratory, Cambridge, Juin 1998.

[150] Jon Postel. Internet protocol, darpa internet program, protocol specification. Ietf request for comment 791, Information Sciences Institute, University of Southern California, Marina del Rey, California 90291, USA, Spetembre 1981.

[151] Jon Postel. Transmission control protocol, darpa internet program, protocol specification. Ietf request for comment 793, Information Sciences Institute, University of Southern California, Marina del Rey, California 902921 USA, Spetembre 1981.

[152] Christian Pérez, Thierry Priol, and André Ribes. A parallel corba component model for numerical code coupling. In Manish Parashar, editor, *Proc. of the 3rd International Workshop on Grid Computing*, number 2536 in LNCS, pages 88–99, Baltimore, Maryland, USA, Novembre 2002. Springer-Verlag. In conjunction with *SuperComputing 2002* (SC'02).

[153] Christian Pérez, Thierry Priol, and André Ribes. An object model for high performance distributed systems. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, Big Island, Hawaii, USA, Janvier 2004. IEEE Computer Society Press. To appear.

[154] Jelica Protić, Milo Tomasević, and Veljko Milutinović. *Distributed Shared Memory: Concepts and Systems*. IEEE, Août 1997.

[155] Niels Provos. libevent. `http://www.libevent.org/`.

[156] L. Prylli and B. Tourancheau. Bip: a new protocol designed for high performance networking on myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*, Lect. Notes in Comp. Science, pages 472–485. Springer-Verlag, apr 1998. In conjunction with *IPPS/SPDP 1998*.

[157] Christophe René and Thierry Priol. MPI code encapsulating using parallel CORBA object. *Cluster Computing*, 3(4):255–263, 2000.

[158] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[159] L. Rivera, L. Zhang, G. Sampemane, and S. Krishnamurthy. HP-RMi: High performance java rmi over fm. Project Report CS 491, Décembre 1997. `http://www-csag.cs.uiuc.edu/individual/achien/cs491-f97/projects/hprmi_paper.ps`.

[160] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local area communication with fast sockets. In *USENIX '97*, pages 257–274, JAN 1997.

[161] Tim Rühl, Henri Bal, Raoul Bhoedjang, Koen Langendoen, and Gregory Benson. Experience with a portability layer for implementing parallel programming systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Sunnyvale, CA USA, AUG 1996.

[162] Robert D. Russel and Phill Hatcher. Efficient kernel support for reliable communication. In *13th ACM Symposium on Applied Computing*, pages 541–550, Atlanta, GA, 1998.

[163] Douglas Schmidt, Aniruddha Gokale, Timothy Harrison, and Guru Parulkar. A high-performance endsystem architecture for real-time CORBA. *IEEE Communication Magazine*, 14(2), Février 1997.

[164] Douglas C. Schmidt. The ADAPTIVE Communication Environment, object-oriented network programming components for developing client/server applications". In *Proceedings of the 12th Annual Sun Users Group Conference (SUG)*, pages 214–225, San Fransisco, CA, Juin 1994.

[165] Douglas C. Schmidt. An architectural overview of the ACE framework: A case-study of successful cross-platform systems software reuse. *USENIX login magazine, Tools special issue*, Novembre 1998.

[166] M. Schulz and S.A. McKee. A framework for portable shared memory programming. In *International Parallel and Distributed Processing Symposium*, page 54a, Nice, France, Avril 2003.

[167] M. Schulz, J. Tao, C. Trinitis, and W. Karl. SMiLE: An integrated, multi–paradigm software infrastructure for sci–based clusters. *Future Generation Computer Systems*, 19(4):521–532, Avril 2003. Special Issue: Selected Papers of CCGrid 2002.

[168] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, Spetembre 1990.

[169] Ashish Singhai. *Quarterware: a Middleware Toolkit of Software RISC Components*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[170] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. Ietf request for comment 1831, Août 1995.

[171] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. SCTP: Stream control transmission protocol. Ietf request for comment 2960, Octobre 2000.

[172] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[173] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[174] Vinod Tipparaju, Jarek Nieplocha, and Dhabaleswar Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, page 84a, Nice, France, Avril 2003.

[175] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open grid services infrastructure (ogsi) version 1.0. Global Grid Forum Draft Recommendation, Juin 2003.

[176] International Telecommunications Union. Network node interface for the synchronous digital hierarchy (SDH). Standard ITU-T G.707, Octobre 2000. `http://www.itu.int/`.

[177] Rob V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Ibis: an efficient java-based grid programming environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, Novembre 2002.

[178] T. von Eicken et al. Active messages: A mechanism for integrated communication and computation. In *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pages 256–155, Los Alamitos, CA, 1992. IEEE Computer Society Press.

[179] John von Neumann. First draft of a report on the edvac. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, Juin 1945. Contract No. W-670-ORD-492.

[180] World Wide Web Consortium (W3C), Octobre 2000. `http://www.w3.org/TR/REC-xml/`.

[181] World Wide Web Consortium (W3C). SOAP version 1.2 part 1: Messaging framework, Juin 2003. `http://www.w3.org/TR/soap12-part1/`.

[182] World Wide Web Consortium (W3C). Web services architecture, working draft 8, Août 2003. `http://www.w3.org/TR/ws-arch/`.

Alexandre DENIS

Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul

Mots-clefs : grille de calcul, parallélisme, systèmes répartis, intergiciels, CORBA, MPI, plate-forme de communication, PadicoTM.

Résumé

L'émergence récente des grilles de calcul donne accès à des ressources à une échelle sans précédent dans l'histoire du calcul scientifique. La question cruciale qui se pose alors est celle de la méthode et du modèle de programmation à employer pour les programmer. Une façon courante de programmer une telle infrastructure est de lui donner l'apparence d'un calculateur parallèle virtuel. Cependant, une telle approche risque de limiter arbitrairement la portée des grilles uniquement aux applications parallèles, alors que d'autres approches comme les objets ou les composants distribués semblent également appropriés à un déploiement sur les grilles. Par conséquent, un environnement logiciel pour la programmation des grilles se doit de supporter les différents modèles de programmation et d'exécution.

Cette thèse étudie un modèle de plate-forme de communication pour la programmation des grilles de calcul. Notre but est d'étendre la portée des grilles en permettant l'exécution d'applications parallèles, réparties, ou une combinaison des deux, sans imposer de contrainte de programmation ou d'exécutif particulier. Le modèle proposé permet l'utilisation d'exécutifs variés tels que MPI, CORBA, SOAP, DSM, JVM, adaptés à l'application plutôt que ceux dictés par les réseaux disponibles. La plate-forme de communication se charge de l'adaptation de chaque exécutif choisi par le concepteur de l'application au-dessus de chaque réseau disponible, et permet toutes les combinaisons exécutif/réseau.

Notre approche est basée sur : un arbitrage des accès aux ressources, pour permettre plusieurs exécutifs simultanément ; une adaptation d'abstraction qui permet de voir tous les réseaux selon le paradigme choisi par l'utilisateur ; une virtualisation des ressources, qui permet l'utilisation d'exécutifs existants sans les modifier. Nous avons mis en oeuvre ce modèle dans la plate-forme PadicoTM, et porté divers exécutifs sur cette plate-forme. Les réseaux utilisables vont des SAN jusqu'aux WAN. Les résultats obtenus montrent que notre approche est compatible avec la haute performance.