Techniques et outils pour les communications et la répartition dynamique de charge dans les réseaux de stations de travail

Olivier Dalle

▶ To cite this version:

Olivier Dalle. Techniques et outils pour les communications et la répartition dynamique de charge dans les réseaux de stations de travail. Calcul parallèle, distribué et partagé [cs.DC]. Université Nice Sophia Antipolis, 1999. Français. NNT : . tel-00712754

HAL Id: tel-00712754
https://theses.hal.science/tel-00712754v1

Submitted on 28 Jun 2012

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.

UNIVERSITÉ DE NICE – SOPHIA ANTIPOLIS
École Doctorale Sciences Pour l'Ingénieur

THÈSE

présentée pour obtenir le titre de

Docteur en Sciences

Discipline : Informatique

par

Olivier DALLE

Techniques et outils pour les communications et la répartition dynamique de charge dans les réseaux de stations de travail

Soutenue le 15 Janvier 1999 à 14h30 devant le jury composé de :

Rapporteurs :	Guy Bernard	Professeur à l'Institut National des Télécommunications
	Franck Cappello	Chargé de Recherche du CNRS (LRI)
	Bertil Folliot	Professeur à l'Université de Paris VI (LIP6)
Examinateurs :	Jean-Claude Bermond	Directeur, Directeur de Recherche du CNRS (I3S)
	Ernst Biersack	Professeur Associé à l'Institut Eurécom
	Michel Syska	Directeur, Maître de Conférences à l'U.N.S.A. (I3S)

à l'École Supérieure en Sciences Informatiques de Sophia Antipolis

Mis en page avec la classe thloria.

Remerciements

Je tiens vivement à remercier les nombreuses personnes qui m'ont aidé, tant sur le plan professionnel que sur le plan personnel, à mener à terme cette longue aventure :

- Mes parents, bien évidemment, pour l'aide et l'affection qu'ils n'ont jamais cessé de m'apporter durant ces longues années ;

- Michel Syska, pour avoir accepté de diriger cette thèse, dans un esprit de franche amitié. Je lui en suis infiniment reconnaissant, non seulement pour l'aide et le temps précieux qu'il m'a consacré, mais aussi pour la confiance qu'il m'a toujours accordé et, surtout, pour la liberté qu'il m'a laissé d'orienter mes travaux dans les directions qui m'intéressaient ;

- Jean-Claude Bermond, pour avoir accepté de co-diriger cette thèse et pour m'avoir accueilli au sein du projet SLOOP. Travailler au sein de son équipe fut incontestablement un grand plaisir. Mais plus encore, je réalise aujourd'hui à quel point ce fut une grande chance pour moi, compte tenu des conditions de travail, de la richesse des formations et des moyens, tant humains que matériels, dont j'ai bénéficié grâce à lui tout au long de cette thèse ;

- Guy Bernard, Franck Cappello et Bertil Folliot pour m'avoir fait l'honneur de participer à mon jury et pour avoir eu la gentillesse et le courage de rapporter cette thèse dans des délais aussi courts ;

- Ernst Biersack, pour m'avoir fait l'honneur de participer à mon jury ;

- Soraya Arias, pour son formidable travail de relecture de ce mémoire. Les corrections que j'ai finalement apportées suite à ses remarques et suggestions sont si nombreuses qu'elle mériterait presque d'y figurer en tant que co-auteur!

- Stéphane Pérennes, d'une part pour son travail de relecture et sa précieuse collaboration lors de la rédaction des parties les plus théoriques de ce manuscrit et, d'autre part, pour les nombreuses et enrichissantes discussions que nous avons eues au long de ces nombreuses années ;

- Nicolas Turro, pour ses patientes relectures et pour m'avoir fait profiter de son expertise dans le domaine des réseaux et de sa maîtrise de certaines techniques que je maîtrisais mal ;

- Philippe Mussi, pour sa générosité, tant d'un point de vue humain que d'un point de vue purement matériel. Au même titre que Jean-Claude Bermond, son soutient constant et sa motivation à me sortir des situations difficiles, à répondre à mes besoins (parfois farfelus), voire même à les devancer, m'ont vraiment permis de d'accomplir cette thèse dans les meilleures conditions ;

- Françoise Baude et Afonso Ferreira, pour leur gentillesse, leur disponibilité et leur collaboration ;

- Ephie Dériche, Fabrice Fenouil, Patricia Lachaume et Zohra Kalafi, tant pour leur efficacité à résoudre mes fréquents embarras administratifs, que pour leur gentillesse et leur disponibilité ;

- Les ingénieurs, chercheurs et personnels techniques de l'INRIA, et en particulier Jean-Paul Bonfils, Stéphane Dalmas, José Grimm, Richard Immordino, Francis Montagnac, Daniel Terrer, Michel Veysset, Franck Yampolski, Antoine Zoggia et les "filles de la doc", pour l'aide technique qu'ils m'ont apporté à maintes occasions ;

- Luc Bougé, Jean-Marc Geib, Jean-François Méhaut, Raymond Namist, Jean-Louis Pazat et, d'une façon générale, l'ensemble des membres du groupe de travail GRAPPES que j'ai eu le plaisir de rencontrer, pour leur accueil sympatique et pour les nombreux enseignements que j'ai tiré de nos réunions ;

- Laurent Villefranche, pour m'avoir supporté avec humour et bonne humeur en tant que co-bureau pendant plusieurs mois ;

- Les doctorants du projet SLOOP, Bruno, David & David, Eric, Gunther, Jean-Noël, Nathalie, Nausica, Olivier, Tania et Yves, pour l'ambiance qu'ils font régner, pratiquement 24 heures sur 24, dans les bureaux et couloirs du projet SLOOP ;

- Les autres membres du projet SLOOP et ceux des équipes voisines (en particulier PACOM et MISTRAL), pour la chaleur de leur accueil et les nombreuses et enrichissantes discussions que nous avons eues ;

- Cyril Godart, pour les nombreuses discussions que nous avons eues et, en particulier, celle qui m'a finalement conduit à concevoir le système MPCFS ;

- Christian Delmas, Nicolas Ebelé et Stéphane Rouvière, pour avoir accepté de relever le défi posé par la conception du tout premier prototype de MPCFS ;

- Cyril, François, Stéphane, Soraya, Nicolas et Peter, mes colocataires successifs, pour leur amitié sincère et les moments inoubliables que j'ai passé avec eux ;

- Et bien-sûr tous mes amis, doctorants, permanents et personnels administratifs à I3S, à l'INRIA et à l'ESSI, avec qui j'ai eu l'occasion de passer d'excellents moments de détente, que ce soit à courir derrière une balle de squash, à jouer au bowling, à faire du VTT, du ski ou simplement à l'occasion d'une soirée : Agnès, Anne, Cédric, Christophe, Colas et ses Koalas, Cyrille, Jean-Christophe, Laurent, Lionel, Monica, Myriam, Nathanael, Olivier, Patrick, Philippe, Régis, Romain, Sandrine, Stéphane, Sylvain, Théo, Vincent, Yves . . .

Table des matières

Table des figures **ix**

1 Introduction **1**
1.1 Stations de travail . . . 2
1.2 Applications réparties . . . 3
1.3 Grappes de stations de travail . . . 4
1.4 Répartition dynamique de charge . . . 5
1.5 Communications . . . 6
1.6 Objectifs . . . 7
1.6.1 Répartition de charge . . . 7
1.6.2 Mécanismes de communication . . . 8
1.7 Plan . . . 9

Partie I État de l'art

2 Indicateurs de charge **15**
2.1 Introduction . . . 16
2.2 Architectures réparties faiblement couplées . . . 17
2.3 Répartition dynamique de charge . . . 22
2.4 Objectif de la répartition dynamique de charge . . . 24
2.5 Informations concernant les processus . . . 27
2.6 Indicateurs de charge . . . 28
2.6.1 Indicateurs de charge mono-dimension . . . 31
2.6.2 Indicateurs de charge multi-dimensions . . . 32
2.7 Prise en compte de l'hétérogénéité . . . 34
2.8 Conclusion . . . 35

3 Communications multipoints **37**
3.1 Introduction . . . 38
3.2 Caractéristiques des communications multipoints . . . 38
3.2.1 Groupes de communication . . . 39
3.2.2 Modes de transmission . . . 41

3.2.3 Schémas de communication 42
3.2.3.1 Schémas de type “un-vers-plusieurs” 42
3.2.3.2 Schémas de type “plusieurs-vers-un” 44
3.2.3.3 Schémas de type “plusieurs-vers-plusieurs” 45
3.3 Protocoles 45
3.3.1 IP/Multicast 46
3.3.2 MTP 46
3.3.3 XTP 47
3.3.4 RAMP 49
3.3.5 RMTP 51
3.4 Environnements de programmation 52
3.4.1 Environnements pour applications tolérantes aux pannes 53
3.4.1.1 Isis 53
3.4.1.2 Transis 54
3.4.1.3 Horus 55
3.4.1.4 Totem 56
3.4.2 Environnements pour applications parallèles 56
3.4.2.1 PVM 56
3.4.2.2 MPI 57
3.5 Systèmes d’exploitation répartis 59
3.5.1 Amoeba 59
3.5.2 Chorus 60
3.6 Conclusion 61

Partie II Contributions

4 Construction d’indicateurs de charge multi-critères en environnement hétérogène 67
4.1 Introduction 68
4.1.1 Motivations 68
4.1.2 Architecture visée 70
4.1.3 Applications ciblées 70
4.1.4 Objectif 71
4.2 Méthodologie 71
4.2.1 Hypothèses 71
4.2.2 Fonction de coût 71
4.2.3 Description de la méthodologie 73
4.2.3.1 Définition des ressources logiques 73
4.2.3.2 Indicateurs de charge 75
4.2.3.3 Étalonnage de la puissance respective de chacun des nœuds 76
4.2.3.4 Modélisation des coûts de communication 76

4.2.3.5 Influence du placement d'une tâche sur les performances des autres tâches . . . 77
4.3 Description de la plate-forme *LoadBuilder* . . . 77
4.3.1 Contraintes . . . 78
4.3.2 Organisation générale . . . 79
4.3.3 Le serveur . . . 80
4.3.4 Le client . . . 82
4.3.5 Modules de charge . . . 83
4.3.5.1 Discussion . . . 83
4.3.5.2 Charge processeur . . . 85
4.3.5.3 Charge mémoire . . . 86
4.3.5.4 Charge réseau . . . 86
4.3.5.5 Charge système . . . 87
4.3.6 Modules d'observation . . . 87
4.3.6.1 Informations locales . . . 87
4.3.6.2 Informations sur le réseau . . . 88
4.3.7 Modules de mesure . . . 89
4.4 Conclusion . . . 92

5 Communications multipoints fiables entre systèmes UNIX 95
5.1 Introduction . . . 96
5.1.1 Plate-forme visée . . . 98
5.1.2 Qualités souhaitées . . . 98
5.1.3 Choix de l'interface de programmation . . . 99
5.2 Présentation de MPCFS . . . 103
5.2.1 Mode d'utilisation . . . 104
5.2.2 Communications multipoints . . . 106
5.2.2.1 Routage . . . 107
5.2.2.2 Primitives d'envoi et de réception . . . 109
5.2.2.3 Caractéristiques des groupes de communication . . . 109
5.3 Communiquer grâce à MPCFS . . . 111
5.3.1 Fonctionnement général, terminologie . . . 111
5.3.2 Arborescence MPCFS . . . 113
5.3.2.1 Point de montage . . . 113
5.3.2.2 Répertoires des groupes de communication . . . 115
5.3.2.3 Les répertoires de multiplexage . . . 116
5.3.2.4 Le lien automatique . . . 118
5.3.2.5 Les nœuds de communication . . . 119
5.3.2.6 Lien vers le message suivant . . . 120
5.3.2.7 Les répertoires de stockage . . . 121
5.3.2.8 Les fichiers contenant les messages . . . 122
5.3.2.9 Le fichier d'enregistrement . . . 123
5.3.2.10 Le fichier d'association . . . 125
5.3.3 Fichiers de configuration . . . 125
5.3.3.1 Configuration initiale . . . 126

5.3.3.2 Définition des groupes 126
5.3.3.3 Le fichier d'export 127
5.3.4 Exemples 128
5.3.4.1 Exemple d'arborescence 128
5.3.4.2 Exemple d'utilisation 130
5.4 Discussion 134
5.5 Conclusion 137

Partie III Mise en oeuvre d'un prototype de MPCFS pour le système Linux

6 Structures de données 141
6.1 Introduction 142
6.2 Gestion des i-nœuds 142
6.3 Groupes de communication 144
6.3.1 Description de la structure 144
6.3.2 Accès aux groupes de communication 145
6.4 Sous-groupes 147
6.5 Membres d'un groupe de communication 149
6.6 Connexions 149
6.6.1 Description de la structure 150
6.6.2 Gestion des files de processus bloqués en écriture 153
6.7 Associations 155
6.7.1 Rôle des associations 155
6.7.2 Contenu de la structure 157
6.8 Messages 158
6.9 Conclusion 159

7 Algorithmes 161
7.1 Introduction 162
7.2 Fonctionnement général 162
7.2.1 Processus de services 163
7.2.2 L'architecture à fils d'exécution multiple du prototype MPCFS . 165
7.2.3 Fonctionnement en mode restreint 166
7.3 Algorithmes de la partie haute 167
7.3.1 Algorithme d'enregistrement 167
7.3.2 Algorithme de dés-enregistrement 169
7.3.3 Émission d'un message 171
7.3.4 Réception d'un message 174
7.3.5 Affiliation d'un processus membre à un sous-groupe 174
7.3.6 Dés-affiliation d'un processus membre d'un sous-groupe 177
7.4 Algorithmes de la partie basse 179
7.4.1 Gestion des *sockets* 179

7.4.2 Traitement d'une connexion active . . . 181
7.4.2.1 Description de l'algorithme de traitement d'une connexion active . . . 181
7.4.2.2 Analyse de l'algorithme de traitement d'une connexion active . . . 183
7.4.3 Associations de machines . . . 185
7.4.4 Dialogue entre les processus utilisateurs et le processus `connd` 186
7.4.5 Établissement des connexions . . . 189
7.4.5.1 Description de l'algorithme d'établissement des connexions . . . 189
7.4.5.2 Gestion de la concurrence . . . 191
7.5 Conclusion . . . 192

8 Protocoles **193**
8.1 Introduction . . . 194
8.2 Couche transport . . . 196
8.2.1 Choix du mécanisme de communication point-à-point . . . 196
8.2.2 Choix du mécanisme de communication multipoints . . . 198
8.2.3 Présentation du protocole de niveau transport . . . 201
8.2.4 Description des champs de l'en-tête réservés au niveau transport 202
8.2.5 Algorithme de traitement d'un message reçu . . . 204
8.3 Couche session . . . 205
8.3.1 Description des champs de l'en-tête réservés au niveau session . 205
8.3.2 Protocole d'envoi des messages utilisateurs . . . 208
8.3.2.1 Envoi d'un message de données . . . 209
8.3.2.2 Réponse positive à un message de données . . . 209
8.3.2.3 Réponse positive à de multiples messages de données 209
8.3.2.4 Réponse négative à un message de données . . . 210
8.3.2.5 Message de reprise des émissions . . . 210
8.3.3 Protocole d'association de groupes . . . 210
8.3.4 Protocole de création de sous-groupes . . . 211
8.3.5 Calcul d'intersection d'arborescences réparties . . . 212
8.3.5.1 Présentation du problème . . . 212
8.3.5.2 Description du protocole . . . 213
8.3.5.3 Analyse de la complexité du protocole . . . 215
8.4 Conclusion . . . 216

---

9 Conclusions et Perspectives **219**
9.1 Construction d'indicateurs de charge . . . 221
9.2 Mécanismes de communication multipoints . . . 223
9.3 Perspectives . . . 225

Annexes

A Guide utilisateur de *LoadBuilder* **227**

A.1 Utilisation du client 228

A.1.1 Format des lignes de commande 228

A.1.2 Syntaxe spécifique de chaque commande 229

A.2 Configuration de l'environnement *LoadBuilder* 232

B Configuration du système de fichier MPCFS **237**

B.1 Fichier d'accès 238

B.2 Fichier d'export 240

Bibliographie **243**

Table des figures

2.1 Noeud d'une architecture répartie 18
2.2 Modèle client/serveur 21
2.3 Exemple de créations de tâches locales et distantes 21
2.4 Exemple de créations de tâches à l'aide d'un service de répartition dynamique de charge 23
2.5 Décroissance comparée des moyennes simples et exponentielles 29

3.1 Exemple d'un séquencement atomique non causal 40
3.2 Étapes d'une transmission de message 41
3.3 Schémas de communication de type "un-vers-plusieurs" 43
3.4 Exemple d'arbre de diffusion sur une grille 44
3.5 Compression des délais inter-paquets dans les routeurs 51

4.1 Exemple de chronologie d'exécution d'une application répartie 72
4.2 Définition et utilisation d'indicateurs de charge multi-dimensions et multi-critères en environnement hétérogène. 74
4.3 Architecture de l'environnement *LoadBuilder* 79
4.4 Exemple de session interactive avec l'environnement *LoadBuilder* 84

5.1 Niveau de mise en œuvre d'un schéma de type *multi-unicast* 97
5.2 Le mécanisme de VFS 102
5.3 Exemple de topologie virtuelle de groupes de communication 108
5.4 Schéma d'organisation des fichiers et répertoires de l'arborescence MPCFS 114
5.5 Exemple d'arborescence MPCFS 129
5.6 Exemple d'une opération de réduction calculant le minimum réparti. 131

6.1 Numérotation des i-nœuds dans le système de fichiers MPCFS. 143
6.2 Accès à la structure d'un groupe de communication 146
6.3 Liens entre sous-groupes 147
6.4 Accès indirect aux messages depuis les connexions 151
6.5 Associations : des structures intermédiaires entre groupes et connexions . . 155
6.6 Accès aux associations à partir de leur identifiant 156
6.7 Accès aux associations à partir d'un groupe ou d'une connexion 156

7.1 Architecture logicielle du système MPCFS 163
7.2 Architecture à fils d'exécution multiples du prototype MPCFS 165

7.3 Structure de message sentinelle utilisée dans les sous-groupes 173
7.4 Réception et traitement d'un message dans le cas idéal. 184
7.5 Établissement d'une connexion MPCFS 190

8.1 Organisation des couches de protocoles dans MPCFS 195
8.2 En-tête des messages MPCFS 203
8.3 Exemple de calcul réparti d'intersection d'arborescence 214

A.1 Exemple de fichier de configuration LoadBuilder 235

Chapitre 1

Introduction

Where a calculator like the ENIAC today is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh only 1 1/2 tons.

–Popular Mechanics, March 1949, p.258

DANS cette thèse, nous proposons des techniques et outils qui permettent une meilleure utilisation des réseaux et grappes de stations de travail, pour l'exécution d'applications réparties.

Avant de décrire les objectifs que nous nous sommes fixés, nous commençons par décrire brièvement le contexte de ce travail, en introduisant les concepts et problèmes auxquels nous nous intéressons. D'abord, nous rappelons comment l'évolution des composants a inévitablement conduit au concept de station de travail et fini par l'imposer, de façon durable. Ensuite, nous introduisons le concept d'application répartie, puis nous présentons la notion de grappe de stations de travail et les traits caractéristiques qui rendent ce type de plate-forme aussi populaire à l'heure actuelle. Puis, nous posons les deux problématiques abordées dans cette thèse : la problématique des communications, et celle de la répartition dynamique de charge. Enfin, nous terminons ce chapitre par une description de l'organisation générale de ce mémoire.

1.1 Stations de travail

En 1965, Gordon Moore, co-fondateur d'Intel, observait que depuis l'invention du circuit intégré, la densité de ces derniers avait doublé tous les ans. Il prédisait de plus, que cette évolution allait se poursuivre à ce rythme longtemps encore, suffisamment pour qu'il n'en voit pas lui-même la fin de son vivant[1]. Bien que le rythme initial se soit par la suite quelque peu ralenti, il est resté suffisamment stable pour donner naissance à la célèbre "Loi de Moore"[2]. D'un point de vue pratique, cette intégration toujours plus grande des composants a non seulement permis de réduire leur encombrement, mais surtout, elle a permis d'accroître régulièrement leurs performances.

Dans les années 80, cette évolution aboutit au concept d'ordinateur individuel. Dans le monde professionnel, ce concept se traduit par l'apparition des *stations de travail*, alors que pour le grand public, il se traduit par l'apparition des micro-ordinateurs. Jusqu'au début des années 90, ces deux branches ont évolué de façon relativement indépendante, en répondant à des objectifs très différents. D'un coté, les stations de travail sont prévues pour fonctionner en *réseau*, avec un *système d'exploitation multi-utilisateurs et multi-tâches* tel que UNIX® . De

1. En 1997, Gordon Moore confirmait sa prédiction, en déclarant que cette évolution devrait se poursuivre jusqu'aux alentours de 2017, où ils se heurteraient à la limite physique infranchissables de la dimension de l'atome.

2. En ce qui concerne les processeurs d'Intel, le doublement de la densité a pu être observé tous les dix-huit à vingt-quatre mois, jusqu'à présent.

l'autre, les micro-ordinateurs, qui doivent rester abordables pour le particulier, sont prévus pour fonctionner de façon autonome, avec, parfois, un système d'exploitation minimal.

Dans les années 90, beaucoup de choses changent. Car si la Loi de Moore prédit une évolution rapide des composants, elle s'accompagne d'un corollaire selon lequel les coûts de développement et de production de ces composants progressent encore plus rapidement. Ces coûts seraient ainsi multipliés par quatre d'une génération de composants à la suivante. Du côté des professionnels, cette explosion des coûts est en partie à l'origine du phénomène de *down-sizing* auquel on assiste. Les gros ordinateurs, trop coûteux au regard des avantages qu'ils peuvent offrir, sont progressivement remplacés par des stations de travail, dont la puissance a par ailleurs nettement progressé. Bien sûr, l'accroissement des coûts n'est pas la seule explication à ce phénomène. Les progrès accomplis en matière de réseaux locaux, par exemple, contribuent pour beaucoup à rendre ce type de configuration viable.

En même temps, les progrès accomplis en matière de composants et de logiciels rendent les micro-ordinateurs de plus en plus attractifs pour le grand public. Cet engouement pour la micro-informatique permet d'amortir les coûts de production et de développement des composants. De plus, les niveaux de performance atteints par ces composants leur permettent techniquement, dès les années 80, de proposer les mêmes services que les stations de travail. Le fossé entre les deux mondes est comblé dans les années 90, avec l'apparition de systèmes UNIX et de logiciels libres et performants pour micro-ordinateurs (Linux, FreeBSD).

La réduction du coût de ces installations leur permet définitivement de s'imposer dans les environnements de recherche et les milieux académiques, où les *réseaux de stations de travail* (*NOW, Networks Of Workstations*) se généralisent. Cette généralisation, en particulier dans les universités, contribue ainsi à faire connaître ce type d'installation à un nombre toujours plus grand d'utilisateurs. Les conséquences de cette vulgarisation sont déjà perceptibles. Dans l'industrie, par exemple, de nombreuses sociétés recourent dorénavant à des stations de travail de type PC sous Linux, pour mettre en place, à moindre frais, des services tels que les serveurs Internet.

1.2 Applications réparties

Définir de façon formelle la notion d'application répartie n'est pas chose facile. Dans ce mémoire, nous appelons une application répartie, *une application qui utilise plusieurs tâches qui coopèrent, prévues pour s'exécuter sur plusieurs machines*. Cette définition est bien sûr discutable. Par exemple, dans [Waldo 94], les auteurs définissent l'application répartie comme une application dont les tâches font appel à des espaces d'adressage différents, éventuellement situés sur des machines différentes. Cette définition qui peut sous-entendre qu'une telle application peut ne pas être prévue pour s'exécuter sur plusieurs machines, ne correspond pas tout-à-fait au type des applications auxquelles nous nous intéressons ici.

On peut distinguer deux types de motivation conduisant à la conception d'applications réparties :

– la dépendance de l'application vis-à-vis de ressources physiquement réparties. Il s'agit dans ce cas d'une répartition par nécessité : l'application est obligée de recourir aux

services ou ressources de plusieurs machines. Typiquement, cette catégorie inclut les *collecticiels* [Cosquer 94], dont la vocation est de supporter plusieurs utilisateurs travaillant, éventuellement de façon simultanée, sur une tâche commune.

– la recherche de performances. Il s'agit dans ce cas d'applications qui exploitent la disponibilité de plusieurs machines pour améliorer leur temps de réponse et/ou leur tolérance aux pannes. Dans le premier cas, on trouve en particulier *les applications parallèles*, dont le principe est de décomposer un traitement coûteux en ressources (processeur, mémoire, disque, etc) en plusieurs parties (données ou algorithmes) pouvant être exécutées de façon concurrente. Dans le deuxième, on trouve, par exemple, des applications ou services qui utilisent la réplication de tâche afin d'augmenter leur tolérance aux pannes et leur disponibilité. Notons que ces deux approches ne sont pas contradictoires.

Les travaux présentés dans cette thèse concernent cette deuxième catégorie d'application, en particulier les applications parallèles. Néanmoins, les solutions proposées peuvent être exploitées par l'ensemble des applications réparties.

1.3 Grappes de stations de travail

Récemment, les performances des réseaux locaux ont fait un bond spectaculaire, en gagnant deux ordres de grandeur après des années de stagnation. La disponibilité de réseaux locaux à très haut débit (1 giga-bit par seconde) et à très faible latence (de l'ordre de quelques micro-secondes, au niveau physique), permet aux réseaux de stations de travail de venir concurrencer les machines parallèles traditionnelles sur le terrain des performances [Cappello 98]. Des réseaux locaux de stations de travail purement dédiés aux calculs parallèles haute performance ont ainsi vu le jour. On les appelle des *grappes de stations de travail* (*Clusters Of Workstations, COW*).

Outre leur rapport performance/prix intéressant, ces grappes présentent en effet de nombreux avantages :

– **Évolutivité** : à l'inverse des machines parallèles où la configuration est généralement assez figée, tous les composants d'une grappe de stations peuvent évoluer, tant sur le plan des performances que sur celui des technologies utilisées. Il suffit de changer l'adaptateur réseau ou le microprocesseur pour mettre la configuration à niveau, dès que de nouveaux composants plus performants apparaissent sur le marché.

– **Extensibilité** : cette propriété est assez limitée dans les machines parallèles. Bien souvent, l'architecture de ces machines ne permet pas d'ajouter des nœuds à l'unité, selon le budget dont on dispose. Par ailleurs, une grappe de stations permet de construire une configuration hétérogène, formée de nœuds de calcul et de composants réseau d'architectures et de générations différentes ;

– **Facilité** : la programmation d'applications réparties sur les grappes de stations de travail, n'exige pas l'apprentissage de nouveaux langages ou de nouveaux systèmes d'exploitation ;

- **Multi-tâches** : cette propriété peut être observée à plusieurs niveaux : au niveau de chacun des nœuds de calcul, au niveau d'un ensemble de nœuds, ou au niveau de l'ensemble de la machine. Les systèmes d'exploitation proposés sur les machines parallèles sont généralement rudimentaires. Ils ne permettent pas toujours un fonctionnement multi-tâches à chacun de ces niveaux. À notre connaissance, tous les réseaux de stations de travail utilisés dans ce contexte sont capables d'assurer un fonctionnement multi-tâches au niveau de chaque nœud de calcul (bien que dans la pratique, leur utilisation soit souvent régie par un système de *batch* tel que LSF [Zhou 92]).
- **Portabilité** : les applications produites pour les réseaux de stations de travail sont moins dépendantes de la plate-forme d'exécution que ne le sont celles produites pour les machines parallèles, qui imposent souvent l'utilisation de bibliothèques et de compilateurs propriétaires.

1.4 Répartition dynamique de charge

La potentiel de ressources inutilisées dans les réseaux d'ordinateurs suscite depuis longtemps l'intérêt de la communauté scientifique [Chou 82, Hwang 82, Barak 85]. L'idée d'exploiter ce potentiel, en particulier au travers d'*algorithmes de répartition dynamique de charge*, n'est donc pas nouvelle. En revanche, la nature de la charge a beaucoup changé. Avec la généralisation des réseaux de stations de travail, l'idée de faire coopérer les machines disponibles pour supporter l'exécution d'applications parallèles s'est rapidement imposée. D'initialement séquentielle, la charge est donc devenue en partie, voire totalement, parallèle.

Or, les caractéristiques des applications parallèles sont bien différentes de celles des applications séquentielles. Tout d'abord, parce qu'elles communiquent beaucoup, et que, comme nous allons le voir au paragraphe suivant, l'efficacité de ces communications doit être préservée. Les stratégies de répartition de charge proposées pour placer les tâches des applications séquentielles sont donc partiellement ou totalement inadéquates, car elles ne tiennent pas compte de ces communications. Ensuite, parce que les communications et le processeur ne sont pas les seuls critères qui doivent être considérés. Les applications parallèles peuvent en effet avoir recours de façon intensive aux entrées/sorties ou être très gourmandes en mémoire.

Bien entendu, le problème de la répartition de la charge des applications parallèles a largement été étudié dans le cadre des machines parallèles [André 88]. Mais ces stratégies doivent aussi être remises en question, car dans ce cas, ce sont les caractéristiques de la plate-forme qui sont très différentes. Les réseaux et grappes de stations de travail sont en effet de nature *fortement hétérogène*. Et cette hétérogénéité est d'autant plus forte qu'elle est très variable, car les machines et les réseaux utilisés évoluent rapidement. Leur configuration change selon les besoins des utilisateurs et les évolutions technologiques.

Les conséquences de ces nouvelles contraintes pour les algorithmes de répartition de charge concernent en particulier leur *politique d'information* : pour répartir équitablement et

efficacement la charge, il faut avant tout être capable de mesurer, et surtout, de comparer cette dernière sur les différentes machines. Or, comme nous venons de le voir, cette charge doit non seulement tenir compte du critère de charge habituellement considéré, c'est-à-dire la charge du processeur, mais elle doit aussi inclure des informations concernant les communications, les entrées/sorties, la mémoire ou toute autre ressource pour laquelle les tâches de l'application sont en concurrence. Ces critères doivent, de plus, permettre d'anticiper les variations de charge afin, si possible, de prévoir l'état de chaque machine au moment où l'information est utilisée [Harchol-Balter 96]. Ils doivent aussi être capables de comparer l'état de charge de machines dont les caractéristiques sont différentes et changeantes. Enfin, et c'est la moindre des choses, ces critères doivent pouvoir être mis en correspondance avec les besoins des tâches de chacune des applications utilisant la plate-forme. La plupart de ces problèmes ont récemment fait l'objet d'études approfondies [Folliot 92, Bernon 95, Chatonnay 98].

1.5 Communications

Les applications réparties dont les tâches sont totalement indépendantes sont rares. Au contraire, leurs tâches ont souvent besoin de communiquer entre-elles, afin d'échanger des données ou pour se synchroniser. Ces phases de communication sont généralement critiques pour les performances de l'application, dont la progression peut être partiellement, voire totalement interrompue, lors des phases de communication. Les mécanismes de communication utilisés doivent donc être aussi efficaces que possible. Cette efficacité peut être mesurée selon deux critères : la fonctionnalité et la performance.

La *fonctionnalité* décrit la capacité du mécanisme de communication à répondre aux besoins des applications. Or, les systèmes d'exploitation UNIX, qui sont présents sur la majorité des stations de travail, ne savent répondre de façon satisfaisante qu'à une trop faible partie de ces besoins. En effet, les seuls mécanismes de communication disponibles sur ces systèmes, sont des mécanismes *conçus pour des communications point-à-point*, même s'ils peuvent en général être détournés pour la réalisation de communications multipoints. Les fonctionnalités proposées par ces mécanismes ne sont donc pas adaptées aux besoins des applications réparties, dont les nombreuses tâches travaillent de façon collective, notamment au travers de *groupes de communication* (les groupes de communication permettent à une tâche de désigner de façon logique un ensemble de tâches réparties). De nombreuses solutions ont été proposées pour répondre à ce problème. Ces solutions consistent soit à palier aux manques du système UNIX, en proposant, par exemple, un environnement ou une bibliothèque de communication telle que PVM [Geist 94], soit à abandonner le système UNIX au profit d'un système réparti tel que Amoeba [Mullender 90], Chorus [Rozier 88] ou MASIX [Card 93]. Ces deux solutions ne sont toutefois pas idéales. La première, qui doit se satisfaire des mécanismes existants pour construire ses fonctionnalités avancées, en contexte d'exécution utilisateur, souffre d'un problème de performance. La seconde résout le problème de façon trop radicale, car il n'est pas toujours souhaitable ou possible d'abandonner le système UNIX au profit d'un autre système.

Sur le plan des *performances*, là encore, le système UNIX est largement mis en défaut, y compris dans le cas de communications point-à-point. En effet, les mécanismes proposés

jusqu'à présent se révélaient suffisant pour des communications point-à-point réalisées sur un réseau aux performances modestes, comme Ethernet à 10 Mbits/s. Avec les réseaux à haut débit et faible latence tels que Myrinet (1 Gbits/s) [Boden 95], les mécanismes proposés par UNIX se révèlent trop coûteux. Pour résoudre ce problème de performance, la solution proposée jusqu'à présent, consiste à fournir aux applications une interface de programmation, ou API (*Application Programming Interface*), qui permette de communiquer directement avec le matériel, sans passer par l'intermédiaire du système d'exploitation. Si cette approche permet effectivement d'atteindre un niveau de performance quasi optimal, comme c'est le cas avec BIP [Prylli 98] ou U-Net [Welsh 97], elle comporte des inconvénients. S'affranchir totalement du système d'exploitation n'est en effet pas toujours désirable. D'une part, cela implique une dépendance de l'application vis-à-vis de l'API de communication proposée, et d'autre part, cela implique souvent de restreindre l'accès à l'interface de communication à un unique processus, au détriment des autres.

1.6 Objectifs

L'objectif du travail présenté dans cette thèse a été de rechercher des solutions aux problèmes de répartition de charge et de réalisation des communications présentés dans les paragraphes précédents.

1.6.1 Répartition de charge

En ce qui concerne la répartition de charge, nous nous sommes intéressés au problème de la définition de critères de charges, dans le contexte particulier des réseaux de stations de travail. Les solutions alors proposées pour résoudre le problème dans ce contexte étaient rares, et certains aspects du problème n'avaient pas encore été traités. Deux de ces aspects ont plus particulièrement retenu notre attention : la nature fortement hétérogène de la plate-forme et son évolution constante. Les conséquences de ces deux caractéristiques sur la définition de critères de charge sont en effet assez lourdes.

L'hétérogénéité implique certes de considérer la différence de puissance des machines, mais cette information n'est, à notre avis, pas suffisante dans un tel contexte. En particulier, il nous apparaît indispensable de considérer aussi les *conséquences* de l'apport d'une nouvelle charge de travail sur chacune des machines. En effet, en environnement homogène, il suffit de comparer la charge des machines pour décider de celle qui est capable d'offrir le meilleur temps de réponse. Cette décision part du principe que, puisque toutes les machines sont identiques, elles réagissent toutes de la même façon à l'apport d'une quantité de charge donnée.

En revanche, en environnement hétérogène, cette supposition ne peut plus être faite. Tout d'abord, l'hétérogénéité de conception des systèmes d'exploitation (Solaris, Linux, Digital UNIX, SGI IRIX, FreeBSD, Windows NT, etc), fait que ceux-ci ne réagissent pas tous de la même façon à une augmentation de la charge. Le type d'architecture (monolithique ou à fils d'exécution multiples), les mécanismes de communication avec le (micro-)noyau (appel de fonction ou passage de messages), les politiques d'ordonnancement, de gestion de

la mémoire virtuelle ou de gestion des caches, sont autant de choix de conception qui ont des conséquences sensibles, lorsque le nombre de processus, l'occupation de la mémoire, la fréquence et le volume des accès aux disques ou le nombre des appels systèmes augmentent [Vahalia 96]. Ensuite, l'hétérogénéité des architectures matérielles a aussi son importance. La taille des mémoires cache des processeurs, par exemple, peut permettre à certains processeurs de supporter la présence d'un nombre de processus plus grand.

De telles différences peuvent ainsi être observées à tous les niveaux : disques, interfaces réseau, mémoire, etc. Savoir prédire les conséquences du placement d'une nouvelle charge de travail sur une machine implique donc d'en étudier le comportement de façon systématique, c'est-à-dire pour tous les niveaux de charges, et non pas seulement avec une charge nulle, comme cela est généralement pratiqué. Mais cette hétérogénéité signifie aussi que les indicateurs habituellement considérés ne sont plus nécessairement suffisants. D'autres indicateurs, comme la fréquence des paginations ou des changements de contexte, par exemple, peuvent fournir des indications complémentaires significatives, que les indicateurs traditionnels (tels que les tailles des files d'attente de processus) ne permettent pas d'observer.

Quant à l'évolution constante des installations, elle implique que cette étude systématique du comportement des machines doit être faite ou refaite chaque fois qu'un élément de la configuration change. Partant de ces observations, nous avons opté pour une approche empirique et automatisée. Nous avons conçu et réalisé un environnement réparti, *LoadBuilder* [Dalle 96], dont l'objectif est de faciliter la réalisation automatique de plans d'expériences. Ces plans d'expériences sont utilisés afin (i) de modéliser le comportement des machines face à la charge et (ii) de rechercher, sur chaque machine, les sources d'informations utiles pour la construction d'indicateurs de charge.

1.6.2 Mécanismes de communication

Alors que le phénomène grappe de stations commençait à prendre de l'ampleur, notamment avec les premières utilisations industrielles conséquentes[3], nous nous sommes intéressé aux mécanismes permettant aux applications réparties de communiquer de façon efficace. En effet, comme nous l'avons remarqué au paragraphe 1.5, les solutions existantes se révèlent souvent trop extrêmes. Aucune des solutions proposées n'a, à notre connaissance, cherché à trouver un compromis entre ces extrêmes. Soit ces solutions proposent des environnements de programmation tels que PVM, certes complets sur le plan fonctionnel, mais pas assez performants car s'exécutant en contexte utilisateur (ce qui devient particulièrement pénalisant avec des réseaux à haut débit). Soit elles consistent à remplacer le système UNIX par un système réparti tel que Amoeba ou Chorus. Mais cette approche n'est pas toujours souhaitable, non seulement pour des raisons de portabilité, mais aussi pour des raisons sociales (imposer un changement de système n'ai pas toujours aisé) [Jia 96]. Soit encore, elles consistent à proposer des mécanismes de communication qui s'affranchissent du système d'exploitation, tel que BIP ou U-Net, mais qui, par la même occasion, perdent les avantages de celui-ci, comme la transparence ou le support multi-tâches et multi-utilisateurs.

3. À titre d'exemple, en 1997, les effets spéciaux du film “Titanic” sont calculés à l'aide d'une grappe de 160 stations DEC alpha sous Linux et Windows NT [Strauss 98].

Nous avons donc cherché une solution intermédiaire, qui permette de combiner au maximum les avantages des solutions existantes, tout en minimisant leurs inconvénients. Clairement, la solution proposée devait se placer au niveau du système d'exploitation, afin d'offrir des performances satisfaisantes. Convaincu de la nécessité de ne pas dépendre d'un système d'exploitation particulier, il fallait donc que cette solution soit compatible avec la majorité des systèmes UNIX existants. Cette solution devait néanmoins offrir les fonctionnalités avancées dont les applications réparties ont besoin. Enfin, il fallait aussi qu'elle permette un fonctionnement multi-utilisateurs et multi-tâches équitable, et donc qu'elle s'intègre à ce système en respectant sa philosophie.

C'est justement cette dernière contrainte qui nous a orientée vers la solution que nous proposons dans cette thèse, le système de fichiers virtuel MPCFS (*Multipoints Communications File System*) [Dalle 98b, Dalle 98a]. En effet, la caractéristique remarquable du système UNIX est son organisation subtile autour de deux paradigmes très simples, les fichiers et les processus. De ce point de vue, le mécanisme de communication multipoints offert par MPCFS respecte parfaitement cette philosophie, mieux encore que les API de communication point-à-point existantes (*sockets, TLI*).

MPCFS permet la construction de *groupes de communications*, au sein desquels les processus d'une application répartie peuvent échanger des messages. Ces groupes sont organisés de façon hiérarchique. Ils apparaissent, dans le système MPCFS, sous la forme d'une arborescence de répertoires. Cette arborescence est reproduite sur chacune des machines, de la même façon qu'un système de fichiers réparti (NFS, par exemple). Les applications trouvent dans cette arborescence des fichiers qui leurs permettent de communiquer (leur fonctionnement est comparable à celui des "tubes nommés" du système UNIX). La programmation de ce système n'exige pas l'utilisation de bibliothèque de communication, puisqu'elle repose sur l'API traditionnelle des systèmes de fichiers (`open(), read(), ...`).

1.7 Plan

Cette thèse se décompose en trois parties. La première est consacrée à un état de l'art, la seconde présente nos contributions, et la troisième décrit la mise en œuvre d'un prototype du système de fichiers MPCFS.

La première partie, consacrée à l'état de l'art, comprend deux chapitres. Le premier est consacré au problème de la définition d'indicateurs de charge pour les algorithmes de répartition dynamique de charge, et le second au problème des communications multi-points.

Dans le premier chapitre, nous présentons les principales techniques connues afin de construire des indicateurs reflétant la charge des nœuds d'une architecture répartie faiblement couplée. Nous commençons donc par décrire les caractéristiques remarquables de ces architectures. Ensuite, nous introduisons le problème général de la répartition dynamique de charge, et posons de façon plus précise celui de la minimisation du temps d'exécution d'une tâche. Puis, nous expliquons comment ce problème conduit naturellement à considérer deux types d'informations : les informations qui décrivent les besoins en ressources des processus, et celles qui décrivent l'utilisation de ces ressources sur chaque machine. Enfin, nous

décrivons les différents indicateurs de charge proposés dans la littérature afin d'évaluer ces dernières.

Dans le second chapitre, nous faisons un large tour d'horizon des solutions proposées au problème des communications multipoints par les différentes communautés intéressées (calcul hautes performances, génie logiciel, systèmes répartis et protocoles). Après avoir introduit les concepts et caractéristiques des communications multipoints, nous décrivons les principales solutions proposées. Nous débutons cette description par les solutions de plus bas niveau, c'est-à-dire les protocoles de communication. Puis, nous abordons les aspects purement logiciels, en présentant les environnements de programmation, issus des recherches en génie logiciel et en parallélisme. Enfin, nous terminons par les aspects "système d'exploitation", en présentant les solutions proposées dans les systèmes d'exploitation répartis.

La deuxième partie, consacrée à nos contributions, est aussi formée de deux chapitres. L'un est consacré à la construction d'indicateurs de charge multi-critères et multi-dimensions, et l'autre aux communications multipoints.

Nous commençons le premier chapitre en explicitant les raisons qui nous ont amené à étudier le problème de la construction d'indicateurs de charge multi-critères et multi-dimensions, puis, nous décrivons la méthodologie de modélisation empirique à laquelle nous avons abouti afin de résoudre ce problème. Enfin, nous présentons *LoadBuilder*, la plate-forme d'expérimentation répartie que nous avons développé afin de mettre cette méthodologie en pratique.

Le second chapitre est consacré au système de fichiers MPCFS. Nous commençons par décrire les contraintes qui ont guidé sa conception. Puis, nous décrivons le système de façon générale, en exposant ses principales propriétés. Nous décrivons ensuite les fonctionnalités proposées par ce système, d'un point de vue utilisation d'abord, et d'un point de vue administration ensuite. Enfin, nous terminons cette présentation en discutant des qualités et défauts de ce système.

La troisième partie décrit les aspects concernant la mise en œuvre du système MPCFS. Nous avons essayé au cours de cette description de nous abstraire autant que faire se pouvait du système choisi pour la mise en œuvre du prototype (Linux), afin notamment d'en démontrer la portabilité sur la majorité des systèmes présents sur les réseaux et grappes de stations de travail.

Dans le premier chapitre, qui peut être lu rapidement lors d'une première lecture de ce document, nous décrivons les principales structures de données utilisées et donnons quelques indications sur leur rôle dans le fonctionnement du système.

Le chapitre suivant décrit et discute les principaux algorithmes que nous avons mis en œuvre dans le prototype. Nous commençons par décrire l'architecture et les principes généraux de fonctionnement du système. Puis nous décrivons les algorithmes de la partie haute du système, s'exécutant au niveau de l'interface avec les applications utilisatrices. Enfin, nous présentons les algorithmes de la partie basse, réalisant l'interface entre les algorithmes précédents et les couches de protocole.

Le dernier chapitre termine la description en présentant les protocoles de communication actuellement utilisés par le prototype. Nous commençons cette description par le protocole de plus bas niveau, dont l'objectif est d'assurer la transmission multipoints fiable des messages (couche transport). Précisons d'ores et déjà que le protocole que nous proposons dans ce prototype n'a pas la prétention d'apporter une solution optimale au difficile problème des communications multipoints fiables. Le seul objectif ayant guidé sa réalisation est, en effet, d'assurer une qualité de service suffisante pour permettre au prototype, dans son ensemble, de fonctionner. Enfin, nous présentons les protocoles de plus haut niveau (couche session), utilisés afin de transmettre les messages utilisateurs au sein de groupes de communications, ou afin d'assurer la gestion répartie de ces groupes de communication.

Finalement, nous concluons cette thèse en faisant un bilan critique de nos contributions et en présentant les perspectives offertes par notre travail.

Première partie

État de l'art

Chapitre 2

Indicateurs de charge

2.1 Introduction

Dans les années 80, l'apparition de réseaux locaux, tels que Ethernet ou Token Ring, a profondément bouleversé la façon d'utiliser les installations informatiques collectives. En rendant possible l'exécution interactive de programmes à distance, en particulier, ces réseaux permettent aux utilisateurs de *choisir dynamiquement* la machine la plus appropriée pour exécuter chacune de leurs commandes interactives, sans avoir à changer de terminal (au travers de commandes telles que `rsh` ou `rlogin`, dans les systèmes UNIX).

Cette possibilité est donc particulièrement intéressante quand les machines sont partagées entre de nombreux utilisateurs, car la charge de travail instantanée de ces machines varie de façon imprévisible. Quand la machine utilisée devient trop chargée, les temps de réponse s'allongent. Avec le mécanisme d'exécution à distance, les utilisateurs qui le désirent peuvent alors tenter leur chance sur une autre machine, sans avoir à se déplacer eux-mêmes physiquement.

Avoir la possibilité de choisir une machine est une chose, faire le bon choix en est une autre. Faire le bon choix signifie choisir une machine qui permet d'améliorer les temps de réponse. Pour cela, plusieurs démarches sont envisageables. La plus simple consiste à choisir une machine au hasard, en espérant qu'elle sera moins chargée. Ce choix est risqué, car il peut aboutir à une machine aussi chargée ou même plus chargée. Cependant, il ne demande qu'un minimum d'effort. A l'opposé, la démarche la plus complexe consiste à étudier chacune des machines de façon rigoureuse, en considérant d'une part des informations connues a priori, comme leur puissance respective ou leur efficacité sur certains traitements (calculs en nombres entiers, calculs en nombres flottants, accès disque, etc), et des informations collectées en temps réel, comme leur nombre d'utilisateurs apparemment actifs, le taux d'occupation de leur(s) processeur(s) et les durées de vie probables des processus en cours d'exécution. Cette démarche prend d'autant plus de temps, que les informations sont nombreuses et/ou difficiles à évaluer. Mais elle peut finalement s'avérer payante, si elle permet systématiquement de réduire le temps de réponse des commandes, de façon significative.

Un utilisateur qui adopte une telle démarche est simplement en train d'appliquer manuellement une stratégie de *répartition dynamique de charge*. Cette démarche est bien sûr trop contraignante pour que l'utilisateur puisse l'appliquer à chacune de ses commandes. Pourtant, les variations de charge sur chaque machine sont souvent très subites : en peu de temps, une machine qui était peu chargée peut devenir très chargée, et inversement. Il est donc naturel de chercher à automatiser cette démarche, afin d'exécuter systématiquement chaque commande sur la machine qui offre potentiellement le meilleur temps de réponse. Toutefois, automatiser la répartition dynamique de charge est un problème compliqué.

En fait, bon nombre des problèmes posés par la répartition dynamique de charge automatique [1] ne sont pas apparus avec les réseaux locaux et les architectures réparties résultantes. En effet, la répartition dynamique de charge est un problème conceptuellement très proche de

1. Par la suite, lorsque nous parlons de "répartition dynamique de charge", il s'agit implicitement de "répartition dynamique de charge automatique".

celui de l'ordonnancement, largement étudié dès les débuts de l'informatique et, bien avant encore, dans le domaine industriel [Coffman 76]. Toutefois, la différence fondamentale entre l'ordonnancement et la répartition de charge est que l'ordonnancement s'intéresse à des problèmes qui peuvent être exprimés en termes de modèles *déterministes*, pour lesquels toutes les informations nécessaires à la résolution du problème, comme le nombre de tâches et leur durée respective, sont connues à l'avance [Gonzalez, Jr. 77]. Au contraire, avec la répartition dynamique de charge, la plupart des informations qui permettraient de trouver un placement optimal des tâches ne sont pas disponibles au moment de la prise de décision.

Pour être efficace, une stratégie de répartition dynamique de charge doit donc chercher à compenser ce manque d'information en choisissant parmi les informations disponibles, celles qui permettent de s'approcher au plus près d'une solution optimale. Ce problème de choix est d'autant plus difficile, que dans une architecture répartie, les informations disponibles sont disséminées. Comme leur temps d'acheminement n'est pas nul, les informations qui sont effectivement disponibles au moment de la prise de décision ne permettent d'avoir qu'une image du système dans le passé [Mirchandaney 89]. Enfin, comme les architectures réparties fonctionnent généralement de façon asynchrone, cette image est souvent incohérente.

L'objectif de ce chapitre est de présenter les différentes approches et solutions proposées dans la littérature, à ce jour, pour résoudre ce problème de choix. Les travaux sur ce sujet, et sur le sujet de la répartition dynamique de charge en général, sont néanmoins trop nombreux pour être tous décrits. Nous allons donc limiter cette présentation à la description d'une sélection de travaux que nous estimons représentatifs, dans le contexte des *architectures réparties faiblement couplées*.

Tout d'abord, nous donnons une description abstraite des architectures réparties faiblement couplées et de leurs caractéristiques remarquables. Nous décrivons ensuite (brièvement) les éléments d'une stratégie de répartition dynamique de charge, afin d'introduire la terminologie utilisée dans ce domaine de recherche[2]. Ensuite, nous présentons l'objectif de la répartition dynamique de charge. Puis, nous décrivons les deux types d'information qui sont utilisées par les algorithmes de répartition dynamique de charge : les informations décrivant les besoins en ressources des processus pris en charge par le mécanisme de répartition de charge, et les information décrivant les ressources actuellement utilisées sur chacune des machines.

2.2 Architectures réparties faiblement couplées

Définition. Une architecture répartie faiblement couplée est formée d'un ensemble de *nœuds* (ou sites) et de *périphériques*, interconnectés par un *réseau de communication*.

Chacun de ces nœuds dispose au minimum d'une unité de traitement (processeur), d'une mémoire volatile propre, et d'une connexion avec le réseau [Tanenbaum 95]. Un nœud peut

2. Le lecteur intéressé par une description plus approfondie de la problématique générale de la répartition de charge et des stratégies existantes est invité à consulter les nombreuses synthèses ayant traité de ce sujet, comme [Wang 85, Casavant 88, Jacqmot 89, Bernard 91b, Folliot 92, Bernon 95, Chatonnay 98].

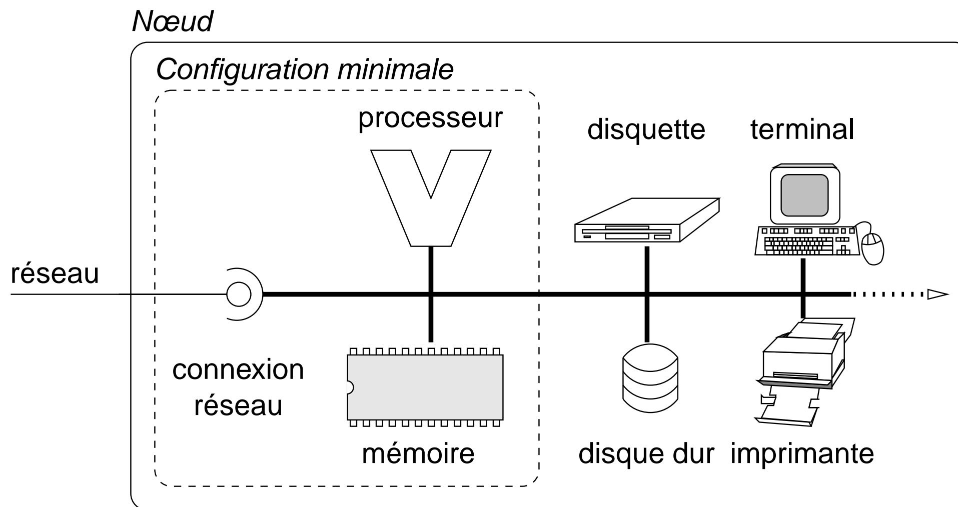


FIG. 2.1: *Noeud d'une architecture répartie*

éventuellement disposer de plusieurs unités de traitement et de ressources additionnelles, comme des périphériques de stockage, des imprimantes, des scanners, des terminaux, etc (figure 2.1). Les réseaux de stations de travail et les machines parallèles MIMD à mémoire répartie sont deux exemples d'architectures réparties faiblement couplées.

Système d'exploitation. La gestion des ressources d'un nœud est prise en charge par un *système d'exploitation* (ou *système opératoire*). Un système d'exploitation peut être mono-tâche ou multi-tâches, selon qu'il permet ou non l'exécution simultanée de plusieurs tâches. Les systèmes d'exploitation mono-tâche ayant progressivement disparu des architectures réparties (ils existaient surtout sur les machines parallèles), nous ne considérons par la suite que des systèmes d'exploitation multi-tâches. Dans les architectures réparties faiblement couplées, le système d'exploitation doit permettre à un nœud d'accéder physiquement aux ressources de chacun des autres nœuds [Bernard 91b]. Un système d'exploitation réparti est *fortement couplé* lorsqu'il gère les ressources de plusieurs nœuds de façon transparente pour ses utilisateurs (comme Amoeba [Mullender 90]) et *faiblement couplé* dans le cas contraire. Par la suite, nous appelons simplement *un système*, soit l'unique nœud pris en charge par un système réparti faiblement couplé, soit l'ensemble des nœuds pris en charge par un système d'exploitation réparti fortement couplé. Enfin, le système d'exploitation peut être mono-utilisateur, comme Windows NT 4.0 [Custer 93] ou multi-utilisateurs, comme UNIX® [Bach 86, Leffler 89].

Réseau de communication. Le réseau de communication permet aux processus d'une application de communiquer soit directement entre eux, soit avec les processus d'autres applications. Les processus d'un nœud sont en concurrence pour l'accès au réseau de la même façon que pour l'accès aux autres ressources. Toutefois, à la différence des autres ressources, l'accès à la ressource de communication se fait en concurrence avec les autres nœuds. Or, pour certains types de réseaux (Ethernet en particulier), l'accès à cette ressource par un nœud

peut être exclusif : lorsqu'un nœud utilise le réseau pour envoyer un message, les autres nœuds sont obligés d'attendre. Par conséquent, les performances du réseau de communication à un instant donné ne sont pas uniquement liées à l'activité des processus qui s'exécutent localement sur un nœud.

Outre cette propriété importante, de nombreuses autres caractéristiques des réseaux ont une influence sur les performances des communications :

- la *taille*, qui définit le nombre de nœuds présents sur un réseau et l'*échelle*, qui définit la dimension du réseau dans l'espace. La terminologie courante distingue habituellement cinq classes de réseau :
  - les *petits* réseaux (*Small Area Networks, SAN*), qui relient des nœuds séparés d'au plus quelques mètres. Ce sont généralement des réseaux à haute performance, comme Myrinet [Boden 95] ou MPC/R-Cube [Zerrouk 96] ou SCI [Gustavson 92] ; ils sont principalement utilisés pour former des grappes de stations de travail ;
  - les réseaux *locaux* (*Local Area Networks, LAN*), tels que Ethernet (10BaseT) ou FastEthernet (100BaseT) [Johnson 96], qui relient des nœuds placés dans un même bâtiment ou des bâtiments voisins. Ils sont principalement utilisés pour former des réseaux et des grappes de stations de travail ;
  - les réseaux *à moyenne échelle* (*Medium Area Networks, MAN*) tels que ATM [Black 95] ou Fast Ethernet sur fibre optique (100BaseFX) [Johnson 96], qui relient des nœuds (ou réseaux) d'un même campus, séparés de quelques centaines de mètres à quelques kilomètres ;
  - les réseaux *à grande échelle* (*Wide Area Networks, WAN*), tel que Internet.
- la *bande passante*, généralement exprimée en bits par secondes, définit la quantité *maximale* d'informations qui peut circuler sur le réseau, par unité de temps. Sur les réseaux locaux, elle est généralement comprise entre 10 MBits/s et 1 GBits/s. Sur un réseau à grande échelle, elle peut n'être que de quelques KBits/s ou atteindre plusieurs centaines de MBits/s. Notons que la bande passante d'un réseau n'est pas toujours constante au cours du temps, en particulier sur des liaisons à grande échelle ;
- la *latence*, exprimée en secondes, milli-secondes ou micro-secondes, définit le délai systématique de prise en charge d'un message. En général, le temps de transmission d'un message peut être représenté par un modèle affine, dans lequel le temps de transmission T d'un message de longueur L est de la forme suivante [Rumeur 94] :

  $$T = \beta + L \cdot \tau \tag{2.1}$$

  Dans ce modèle, le temps β correspond au temps de latence et le coefficient τ à l'inverse de la bande passante ;
- le *débit*, généralement exprimé en bits par seconde, définit la quantité d'information effectivement transmise par unité de temps. Lorsque le temps de transmission des messages respecte le modèle linéaire de la relation 2.1, le débit (noté ρ) dépend de la

longueur des messages transmis :

$$\rho(L) = \frac{L}{T} = \frac{L}{\beta + L \cdot \tau} \; ; \tag{2.2}$$

– la *fiabilité* indique si un message confié au réseau peut se perdre en cours de transmission ;

– la *capacité d'adressage multipoints* indique si un même message peut être envoyé en une seule transmission vers plusieurs destinataires. La *diffusion* est un cas particulier d'adressage multipoints dans lequel un message peut être envoyé à l'ensemble des nœuds du réseau en une unique transmission ;

– la *topologie* décrit l'organisation des liens qui forment le réseau. Les topologies des réseaux de stations de travail sont généralement des combinaisons d'étoiles (routeurs, ponts), de bus (répéteurs Ethernet) et d'anneaux (Token Ring, FDDI). Les machines parallèles et réseaux à hautes performances peuvent avoir des topologies plus élaborées, comme la grille ou l'hypercube, ou même des topologies reconfigurables. La topologie introduit la notion de *distance* entre nœuds (le nombre de liens différents empruntés) et de *degré* d'un nœud (nombre de voisins directs) ; elle a des conséquences importantes sur les temps de transmission et le comportement du réseau en cas de forte charge [Rumeur 94].

Hétérogénéité. L'hétérogénéité est l'une des principales caractéristiques des architectures réparties faiblement couplées, où elle apparaît à tous les niveaux [Zhou 93, Bernard 96] :

– *Hétérogénéité des architectures de machines* : elle concerne en premier lieu les processeurs (Intel x86, DEC Aplha, SPARC, MIPS, HP-PA, PowerPC) dont les caractéristiques et les performances sont très variables [Seznec 96, Seznec 97], y compris au sein d'un même gamme de processeurs, dont les fréquences d'horloge ne cessent de croître. Mais elle concerne aussi les bus d'adressage (PCI, SBUS, VME, etc), les gestionnaires de mémoire cache (nombre de niveaux, type de cohérence), ou le type des mémoires utilisées (temps d'accès, type de rafraîchissement, etc).

– *Hétérogénéité des systèmes d'exploitation* : le fonctionnement et l'architecture des systèmes d'exploitation sont très variables [Vahalia 96]. Certains systèmes, comme Solaris, ont un noyau à fils d'exécution multiple. Certains autres, comme Windows NT 4.0, sont multi-tâches mais ne sont pas multi-utilisateurs. D'autres encore, comme Digital UNIX (anciennement OSF/1) sont construits autour d'un micro-noyau tel que Mach. D'autres enfin comme Linux, sont bâtis de façon très classique [Bach 86], autour d'un noyau monolithique.

– *Hétérogénéité des réseaux* : elle apparaît au niveau de chacune des caractéristiques que nous avons recensé au paragraphe précédent. Ses conséquences s'observent donc tant au niveau des performances, qui varient de 10 MBits/s pour un réseau Ethernet classique à 1 GBits/s pour un réseau Myrinet, qu'au niveau de leur mode d'utilisation

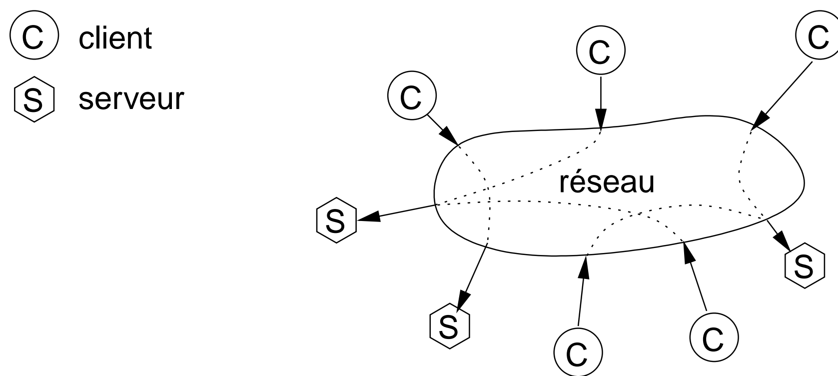


FIG. 2.2: *Modèle client/serveur*

(capacité de diffusion ou de multi-distribution, commutation de circuit, commutation de paquets, capacité full-duplex ou half-duplex, etc) ;

– *Hétérogénéité des configurations* : deux machines attachées sur un même réseau, ayant des processeurs et un système d'exploitation identiques, peuvent néanmoins être très différentes, selon la quantité de mémoire dont elles disposent, selon le nombre et le type de leurs (éventuels) disques durs, ou les caractéristiques de leur interface réseau.

Modèle client/serveur. Un nœud qui propose un service à d'autres nœuds est appelé un *serveur*. Un nœud qui a recours aux services d'un serveur est appelé un *client* (figure 2.2). Ces deux fonctions ne sont pas exclusives : un nœud peut jouer à la fois les rôles de client et de serveur. La nature et le nombre des services proposés par les serveurs sont très variés. Les deux plus courants sont l'exécution distante et le partage de fichiers. Dans le premier cas, on parle de *serveur de calcul* ou de *nœud de calcul*, et dans le second de *serveur de fichiers*.

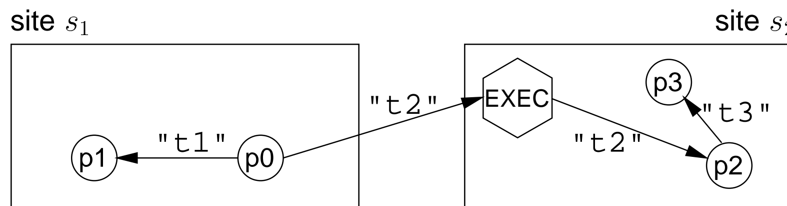


FIG. 2.3: *Exemple de créations de tâches locales et distantes*

Tâches, programmes et processus. Un *programme* représente une suite d'instructions rédigées dans un langage particulier et utilisées par l'ordinateur pour opérer un traitement. Le traitement réalisé par un programme peut être précisé au moment de son invocation, au travers de paramètres d'exécution. Une *tâche* désigne un programme et ses éventuels paramètres d'exécution. Un *processus* représente une tâche en cours d'exécution.

La figure 2.3 représente deux serveurs de calcul, s_1 et s_2. Le processus p0, qui s'exécute sur le serveur s_1, demande à son propre site d'exécuter la tâche `"t1"` et demande au site

s_2 d'exécuter la tâche "t2". Ces demandes aboutissent respectivement à la création des processus p1 et p2. Sur le site s_2, le processus p2, qui exécute la tâche "t2" demande à son propre site d'exécuter la tâche "t3", ce qui aboutit à la création du processus p3.

2.3 Répartition dynamique de charge

Charge. La *charge* d'un site à un instant donné représente l'ensemble des demandes en ressources des processus locaux et du système opératoire de ce site. Les demandes en ressources sur chacun des sites sont très variables au cours du temps, tant par leur type que par la quantité de ressource demandée. Cette variabilité conduit donc à des déséquilibres importants entre sites.

Répartition de charge. La *répartition de charge* désigne la technique générale qui consiste à corriger ces déséquilibres en déplaçant les processus depuis les sites les plus chargés vers les sites les moins chargés. Lorsque les besoins en ressources des applications sont connus à l'avance et lorsque la charge dûe aux autres applications peut être négligée (en environnement mono-utilisateur, par exemple), cette répartition de charge peut être réalisée de façon statique [André 88]. Mais le plus souvent, comme l'une et/ou l'autre de ces deux conditions ne peuvent être vérifiées, cette répartition doit être réalisée en cours d'exécution. On parle alors de *répartition dynamique de charge*. De nombreuses stratégies ont été proposées pour répartir dynamiquement la charge, comme par exemple [Barak 85, Litzkow 88, Kunz 91, Bernard 91a, Boutaba 92, Guyennet 93, Chatonnay 98]. Ces stratégies se distinguent selon le type d'information qu'elles utilisent pour évaluer la charge des sites, selon qu'elles s'exécutent de façon centralisée ou répartie, selon qu'elles sont préemptive ou non, selon qu'elles utilisent tout ou partie de l'information disponible, selon que la répartition est déclenchée à l'initiative des sites les plus chargés ou des sites les moins chargés, etc [Wang 85, Casavant 88, Jacqmot 89, Bernard 91b].

La figure 2.4 illustre le fonctionnement d'une stratégie de répartition dynamique de charge, en reprenant le scénario utilisé pour la figure 2.3. Le processus p0 qui s'exécute sur le site s_1 demande au service de répartition dynamique de charge de créer les deux tâches "t1" et "t2". La stratégie de répartition choisi les deux sites s_i et s_j pour exécuter ces deux tâches et lance les processus correspondants, p1 et p2. Sur s_i, p2 demande à son tour l'exécution d'une nouvelle tâche, "t3". La stratégie désigne cette fois le site s_k, sur lequel le processus p3 est créé.

Éléments d'une stratégie de répartition dynamique de charge. Les stratégies de répartition dynamique de charge sont construites autour de deux éléments [Mehra 93] : les *métriques de charge*, qui reflètent l'état de charge de chaque site, et les *politiques de décision*, qui définissent les règles d'utilisation de ces métriques et les modalités de réalisation de la répartition dynamique de charge. Un algorithme typique de répartition dynamique de charge combine les trois politiques suivantes [Bernard 91b, Talbi 95] : une *politique d'information*, qui définit quelles informations doivent être utilisées et comment elles doivent être collectées, une *politique de transfert* dont le rôle est de déterminer à quel moment un pro-

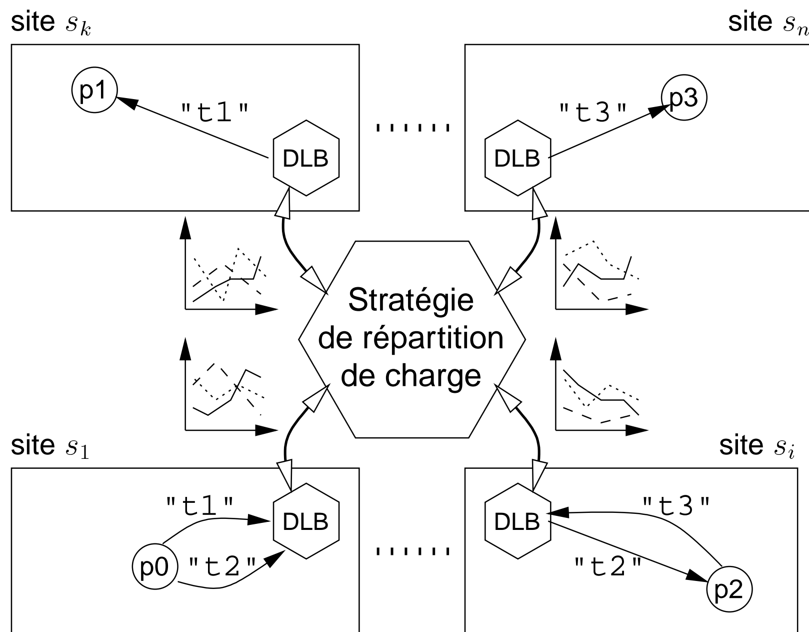


FIG. 2.4: *Exemple de créations de tâches à l'aide d'un service de répartition dynamique de charge*

cessus doit être déplacé et lequel doit être choisi parmi l'ensemble des processus éligibles[3], et une *politique de localisation*, dont le rôle est de choisir vers quel site un processus doit être déplacé. Il existe deux catégories de politiques de transfert : les *politiques de migration* et les *politiques de placement*. Les politiques de migration s'autorisent le déplacement d'un processus en cours d'exécution alors que les politiques de placement ne s'intéressent aux processus qu'au moment de leur création.

Aspects temporels. Le *temps d'exécution d'un processus* est le temps qui s'écoule entre le moment ou le processus débute son exécution et celui où il termine son exécution. Le *temps d'exécution d'une tâche* est le temps qui s'écoule entre le moment où l'application (un de ses processus) demande l'exécution de cette tâche et le moment où cette tâche est complétée. Le *temps de lancement d'une tâche* est le temps qui s'écoule entre le moment où l'application demande l'exécution d'une tâche et le moment où le processus chargé de réaliser cette tâche débute son exécution[4]. Le temps d'exécution d'une tâche est donc la somme de son temps de lancement et du temps d'exécution du processus chargé de sa réalisation.

3. Ces deux aspect sont parfois séparés en deux politiques différentes. Par exemple, [Fonlupt 95] les appelle respectivement *politique de déclenchement* et *politique de désignation locale*.

4. Dans le cas d'une application répartie, comme les tâches peuvent s'exécuter sur des machines différentes, le temps de lancement d'une tâche n'est pas négligeable.

2.4 Objectif de la répartition dynamique de charge

Pour [Wang 85], les propriétés désirables d'un algorithme de répartition de charge sont les suivantes :

1. performance globale du système optimale : la capacité de calcul totale de la plate-forme doit être maximisée et conduire à des temps de réponse acceptables ;
2. équité du service : toutes les tâches prises en charge doivent bénéficier d'une performance uniformément acceptable ;
3. tolérance aux pannes : la robustesse et la performance du système doivent être garanties en présence de pannes partielles du système.

En supposant qu'ils sont classés par ordre de priorité (ce qui n'est pas précisé par les auteurs), ces objectifs s'inscrivent donc plutôt dans l'optique d'une optimisation de l'utilisation globale des ressources du système. En particulier, la première propriété n'accorde qu'une priorité relative à l'optimisation des performances des applications, c'est-à-dire la minimisation de leur temps total d'exécution. Or, dans un contexte hétérogène, ces deux objectifs sont parfois antagonistes. Supposons, par exemple, que la plate-forme ne comporte que deux nœuds, X_{10} et Y_1, tels que X_{10} soit 10 fois plus puissant que Y_1. Supposons, de plus, que Y_1 soit inactif et que quatre tâches soient déjà en cours d'exécution sur X_{10}. Supposons, maintenant, qu'une nouvelle tâche se présente dans le système. La première propriété implique de confier cette tâche au nœud Y_1, car de cette façon, la capacité de calcul délivrée par le système est maximale. En revanche, si elle est placée sur X_{10}, cette tâche bénéficie d'un cinquième de ses performances, c'est-à-dire du double de ce que peut offrir Y_1. Bien sûr, la deuxième des propriétés désirées tend à s'opposer à ce phénomène, car elle implique de minimiser les différences de performance. Mais comme ces deux propriétés sont antagonistes, leur priorité respective doit être clairement précisée.

Dans le contexte actuel, bon nombre d'utilisateurs s'intéressent aux architectures réparties afin d'optimiser le temps de réponse de leurs applications les plus gourmandes, et en particulier de leurs applications parallèles ou réparties. Dans ce contexte, il nous paraît donc plus judicieux d'accorder la plus forte priorité à l'objectif de minimisation du temps de réponse de ces applications, plutôt qu'à celui de la maximisation de la capacité de calcul du système.

Le gain de performance apporté par la répartition de charge est mesuré en terme d'accélération (*speed-up*), par rapport à une exécution locale sans répartition de charge (tous les processus de l'application sont exécutés sur la même machine), ou par rapport à une stratégie de répartition rudimentaire ne tenant pas compte de l'information de charge, comme le placement cyclique (*round-robin*) ou le placement aléatoire [Mehra 93, Barak 98]. Notons que selon le contexte, l'objectif peut être d'améliorer les performances dans le pire des cas (contexte temps réel) ou en moyenne (contexte général).

Comme nous l'avons vu en introduction de ce chapitre, le problème du placement optimal ne peut pas être résolu de façon déterministe, car les informations qui permettraient

d'atteindre une solution optimale ne sont pas disponibles. Cet objectif ne peut donc être atteint que par l'utilisation d'heuristiques permettant de réduire, autant que faire se peut, le temps d'exécution des applications. Les heuristiques proposées découlent en général de règles de bon sens, et en particulier des deux suivantes :

1. le temps d'exécution de l'application peut être réduit si l'on évite de gaspiller des ressources ;
2. il faut essayer de trouver pour chaque tâche, un site d'exécution qui minimise son temps d'exécution.

Toutefois, appliquer ces deux règles de bon sens n'est pas toujours aussi simple qu'il n'y paraît. Pour appliquer la première, par exemple, une solution simple, souvent proposée, consiste à construire un algorithme qui recherche les stations inutilisées [Litzkow 88, Cap 93, Arpaci 95]. Or, comme nous venons de le voir, dans un contexte fortement hétérogène, préférer une machine inutilisée mais de faible puissance peut s'avérer moins efficace que de choisir une machine chargée mais puissante. Remarquons de plus, que dans un tel contexte hétérogène, les risques de se trouver dans cette situation, et donc de faire ce "mauvais" choix, sont d'autant plus importants que les utilisateurs ont naturellement tendance à préférer les machines puissantes aux machines peu puissantes. Les machines peu puissantes sont donc généralement plus disponibles. Pourtant, cette stratégie présente aussi des avantages. Tout d'abord, les informations requises sont très simples à évaluer. Ensuite, elle permet facilement de mettre en place une stratégie dans laquelle on s'interdit de déranger les utilisateurs "propriétaires" des stations. Enfin, si une machine puissante est chargée, cela signifie que d'autres tâches sont en train d'être exécutées sur cette machine. En ajouter une autre pénalise donc l'exécution de ces tâches, alors que le choix d'une machine inactive ne pénalise aucune tâche.

Minimiser le temps d'exécution d'une application est un problème d'optimisation très complexe, en particulier lorsque l'application est multi-tâches (répartie) et que le contexte est hétérogène. Le placement d'applications mono-tâche en environnement hétérogène est à lui seul est un problème très difficile, car il comporte un grand nombre de paramètres. Dans le cas d'une application multi-tâches, il faut de plus prendre en compte les effets du placement d'une tâche de l'application sur un site où s'exécutent déjà une ou plusieurs tâches de l'application.

Afin d'exhiber ces paramètres, nous allons présenter le problème de façon plus formelle. Toutefois, pour ne pas alourdir inutilement cette présentation, nous allons nous limiter à la présentation du problème dans le cas d'applications mono-tâche.

Soient :

– $T(w, s_i, s_j)$, le temps d'exécution depuis un site s_i d'une tâche w sur un site s_j [5] ;
– $T_l(s_i, s_j)$, le temps de lancement d'une tâche quelconque depuis le site s_i vers le site s_j ;

5. Les tâches ne sont jamais créées de façon spontanée, mais résultent d'une demande de création émise par un processus sur un site. s_i désigne donc le site sur lequel s'exécute le processus ayant émis la demande de création de la tâche w.

– $T_e(w,s)$, le temps d'exécution du processus réalisant la tâche w sur le site s ;

– $T_d(w,s)$, le temps nécessaire à l'algorithme pour décider sur quel site exécuter une tâche w dont la création a été demandée à partir du site s ;

– $T_s(w,s_i,s_j)$, le sur-coût lié à l'exécution de la tâche w sur le s_j plutôt que sur le site s_i ;

– $\mathcal{R}_j(w) = \{r_{1_j} \cdots r_{n_j}\}$, l'ensemble des ressources utilisées par le processus réalisant la tâche w sur le site s_j [6] ;

– $U_j(r,w)$, le temps d'utilisation par le processus réalisant la tâche w sur le site s_j de la ressource $r \in \mathcal{R}_j(w)$;

– $A_j(r,w)$, le temps d'attente par le processus réalisant la tâche w sur le site s_j de la ressource $r \in \mathcal{R}_j(w)$;

– $\mathcal{I}_j(w)$, le temps pendant lequel le processus est inactif, car bloqué dans l'attente d'un événement (clavier, souris, message, etc).

Posons

$$\mathcal{U}_j(w) = \sum_{k=1_j}^{u_j} U_j(r_k,w) \text{ et}$$
$$\mathcal{A}_j(w) = \sum_{k=1_j}^{u_j} A_j(r_k,w), \text{ avec } r_k \in \mathcal{R}_j(w).$$

Alors, on a :

$$T(w,s_i,s_j) = T_l(s_i,s_j) + T_e(w,s_j) \tag{2.3}$$
$$T_l(s_i,s_j) = \begin{cases} T_d(w,s_i) & \text{si } i = j, \\ T_d(w,s_i) + T_s(w,s_i,s_j) & \text{si } i \neq j. \end{cases} \tag{2.4}$$
$$T_e(w,s_j) = \mathcal{I}_j(w) + \mathcal{U}_j(w) + \mathcal{A}_j(w). \tag{2.5}$$

Minimiser le temps d'exécution d'une tâche w lancée depuis un processus s'exécutant sur le site s_i signifie : rechercher un site s_j tel que $T(w,s_i,s_j)$ soit minimal. Notons que lorsque les informations permettant à l'algorithme de prendre sa décision sont immédiatement disponibles, le temps de décision $T_d(w,s)$ peut être considéré négligeable. Par ailleurs, lorsque le temps de décision $T_d(w,s)$ est indépendant du site s (et en particulier lorsque il est nul), l'égalité 2.4 implique que pour qu'il soit intéressant d'exécuter une tâche w sur un site s_j différent du site s_i (le site du processus qui demande l'exécution de w), il faut que :

$$T_e(w,s_j) - T_e(w,s_i) > T_s(w,s_i,s_j). \tag{2.6}$$

6. Le double niveau d'indexation des ressources signifie que le nombre et le type des ressources utilisées peuvent varier d'un site à un l'autre.

Généralement, le temps de décision $T_d(w, s)$ et le sur-coût $T_s(w, s_i, s_j)$ peuvent être majorés ou négligés. Toute la difficulté de ce problème d'optimisation réside donc dans l'évaluation de la quantité $T_e(w, s)$ décrite par l'égalité 2.5[7]. En premier lieu, ce temps dépend bien sûr de l'*intensité des activités du processus*, c'est-à-dire le ratio du temps d'inactivité sur le temps total d'exécution[8]. Par exemple, les applications interactives sont des applications de faible intensité alors que les applications de calcul sont des applications de forte intensité. En second lieu, ce temps dépend de la quantité de ressources $\mathcal{U}_j(w)$ dont a besoin le processus au cours de son exécution. En troisième lieu, ce temps dépend de l'état de charge $\mathcal{A}_j(w)$ du site s_j, c'est-à-dire du temps que le processus passe à attendre les ressources dont il a besoin [Casavant 88].

2.5 Informations concernant les processus

Pour évaluer la quantité $\mathcal{U}_j(w)$, les stratégies de répartition de charge doivent donc disposer d'informations concernant les processus qu'elles prennent en charge. Ces informations peuvent être classées en deux catégories : d'un côté les *informations a priori*, qui sont obtenues avant l'exécution du processus, et de l'autre les *informations dynamiques*, qui sont obtenues pendant l'exécution du processus.

Informations a priori. Les informations a priori sont des indications données par l'application lors du lancement de l'un de ses processus ou des indications concernant le comportement des processus, enregistrées lors d'exécutions précédentes de l'application [Hémery 94]. Par exemple, ces informations peuvent être les suivantes :

- la durée d'exécution estimée ;
- une description des liaisons (communications) avec les autres processus ;
- le temps moyen et l'intensité d'utilisation du processeur ;
- la quantité de mémoire utilisée ;
- une description des entrée sorties réalisées ;

Les informations données par l'application peuvent aussi inclure des contraintes sur le placement. Ces contraintes peuvent concerner le matériel requis pour l'exécution d'une tâche ou une liste de processus avec lesquels un processus désire être (ou ne pas être) placé [Folliot 92, Bernon 95, Chatonnay 98].

---

7. L'égalité 2.5 est une version simplifiée du modèle proposé et démontré dans [Ferrari 86] (dans le cas homogène).

8. Cette quantité est aussi appelé *poids* du processus par certains auteurs [Bernon 95].

Informations dynamiques. Il s'agit, par exemple, de la durée d'exécution du processus au moment de l'observation, des ressources utilisées par le processus au moment de l'observation, ou de l'historique des ressources utilisées par le processus.

Les informations dynamiques permettent en particulier de concevoir des stratégies de répartition de charge qui fonctionnent de façon totalement transparente pour les applications, c'est-à-dire sans exiger la collaboration de ces dernières. Cette propriété est particulièrement intéressante lorsque le mécanisme de répartition est placé au niveau du système opératoire, car elle permet la prise en charge systématique de tous les processus.

Lorsque le temps total d'exécution d'un processus n'est pas connu a priori, la durée d'exécution d'un processus au moment de l'observation, par exemple, est une information très utile. En effet, comme le révèle [Cabrera 86], la majorité des processus d'un système UNIX ont une durée de vie très inférieure à la seconde. Son étude révèle, par exemple, que 70 à 80 pourcent des processus ont une durée de vie inférieure à 400 milli-secondes. A l'opposé, cette étude, ainsi que [Leland 86] et [Harchol-Balter 96], révèlent aussi que plus un processus a vécu longtemps, plus la probabilité qu'il vive encore longtemps est importante[9].

Domaine d'utilisation. Dans un contexte hétérogène, les deux seules solutions qui sont assurées de fonctionner systématiquement sont la migration avec la coopération de l'application [Cabillic 96], et le placement. Dans les deux cas, les stratégies de répartition doivent donc, si possible, disposer d'informations a priori concernant les processus. Lorsque ce n'est pas le cas, les stratégies proposées jusqu'à présent se contentent de considérer que la seule ressource utilisée est le processeur.

2.6 Indicateurs de charge

Les indicateurs de charge sont utilisés pour évaluer l'état de charge d'un site (la quantité $\mathcal{A}_j(w)$ dans la relation 2.5) [Ferrari 86, Ferrari 88, Berry 91, Calzarossa 93].

Files d'attente. De nombreuses métriques expriment la charge en mesurant la taille des files d'attente sur lesquelles sont placés les processus dans l'attente d'une ressource. En effet, comme nous le rappelons avec la relation 2.5, la différence entre le temps d'exécution $T_e(w, s_j)$ d'un processus w qui n'est en concurrence avec aucun autre processus sur un site s_j et le temps d'exécution de ce même processus lorsqu'il est concurrence avec d'autres processus sur le même site s_j pour l'accès aux ressources $\mathcal{R}_j(w)$ dont il a besoin est : $\mathcal{A}_j(w) = \sum_{k=1_j}^{u_j} A_j(r_k, w)$ où $\{r_k, k = 1_j \cdots n_j\} = \mathcal{R}_j(w)$.

[Ferrari 86] précisent cette relation en démontrant que $\mathcal{A}_j(r_k, w)$ est une combinaison linéaire de la taille moyenne des files d'attente pour chacune des ressources utilisées par le

9. Plus précisément, l'étude de Cabrera montre que dans le cas de processus à durée de vie longue (supérieure à 500 milli-secondes), lorsqu'un processus s'est exécuté pendant un laps de temps T et qu'il est toujours en vie, alors il y a 40 pourcent de chances pour que sa durée de vie atteigne 2T.

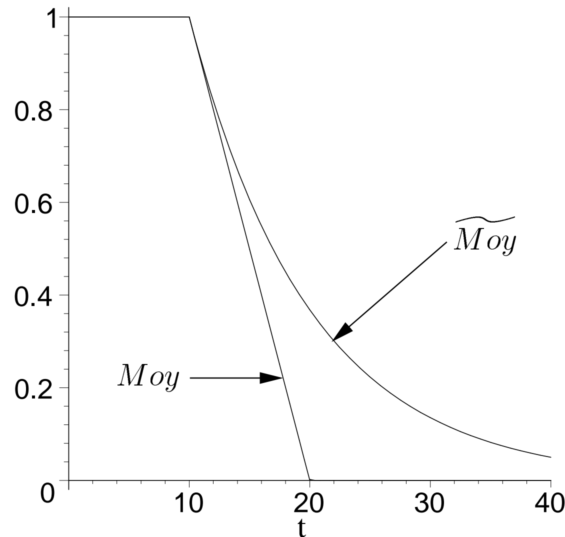


FIG. 2.5: *Décroissance comparée des moyennes simples et exponentielles*

processus w lorsque la machine subit la charge d'arrière plan. Les coefficients de cette combinaison linéaire représentent le temps que passerait le processus w à utiliser chacune de ces ressources s'il était seul sur la machine. La charge d'arrière plan est la charge moyenne dûe à l'exécution des autres processus. Évidemment, cette dernière charge n'est pas connue avant l'exécution du processus. Mais dès lors que l'on suppose l'état de charge du site suffisamment stable, et/ou que l'on est capable d'en prédire l'évolution future (d'après les évolutions récentes), il devient possible d'estimer le temps d'exécution du processus w sur le site s_j. Toutefois, comme le précise encore cette étude, cette estimation de la charge future est d'autant plus difficile que la durée d'exécution de la tâche w est longue.

Moyennes. Pour compenser ces variations, plusieurs méthodes ont été proposées afin de construire des indicateurs à partir de la longueur *moyenne* des files d'attente sur un intervalle de temps T. Les deux méthodes de calcul de la moyenne généralement utilisées (de façon séparée ou combinée) sont les suivantes :

1. *la moyenne simple*, que nous notons $Moy(Q, q, T, t)$. Le calcul de cette moyenne à la date t est obtenu à partir d'échantillons Q_i de la valeur instantanée de la quantité Q, collectés à intervalles de temps réguliers q (appelé le *quantum*) sur l'intervalle de temps $[t - T; t]$. En prenant $\mu \in \mathbb{N}$ tel que $T = \mu \cdot q$, nous avons :

$$Moy(Q, q, T, t) = \frac{1}{\mu} \cdot \sum_{i=1}^{\mu} Q_i \tag{2.7}$$

2. *la moyenne amortie exponentiellement*, que nous notons $\widetilde{Moy}(Q, q, \lambda, t)$. Cette

moyenne est calculée de façon itérative :

$$\widetilde{Moy}(Q, q, \lambda, t) = \widetilde{Moy}(Q, q, \lambda, t-1) \cdot e^{-\lambda} + Q_i \cdot (1 - e^{-\lambda}) \tag{2.8}$$

Comme précédemment, les paramètres Q, q et t représentent respectivement la quantité mesurée, le quantum de temps suivant lequel cette quantité est échantillonnée et la date du calcul. Le principe de l'amortissement exponentiel est de faire en sorte que le poids relatif d'un échantillon dans la moyenne décroisse de façon exponentielle au cours du temps, selon une loi de type $p(t) = e^{-\lambda \cdot t} \cdot p(t_0)$. Le paramètre λ, qui doit être choisi positif, permet de régler la vitesse de dépréciation des échantillons. Ce paramètre est (en principe) calculé de telle façon qu'après un temps t_c, le poids $p(t_c)$ soit inférieur à une valeur seuil p_c. Par analogie avec la moyenne simple, la durée t_c représente donc l'intervalle de temps sur lequel la moyenne est calculée.

La figure 2.5 illustre le comportement de ces deux moyennes sur un exemple simple : l'indicateur reste constant à 1.0 jusqu'à $t = 10s$ où il chute brutalement à 0.0. Les deux moyennes sont calculées sur un intervalle de 10s ($p_c = e^{-1}$).

Mise en pratique. Idéalement, pour construire un indicateur de charge capable d'estimer le temps d'exécution d'un processus w sur un site s_j, il faut disposer :

– de la description quantitative des ressources $\mathcal{R}_j(w)$ dont a besoin le processus w sur le site s_j. Cette description peut (éventuellement) provenir des informations décrites au paragraphe 2.5 concernant les processus ;

– de la description (prédiction) quantitative des ressources $\mathcal{R}_j(w)$ qui vont être utilisées par les autres processus pendant l'exécution du processus w sur le site s_j.

En pratique, ces deux descriptions peuvent ne pas être disponibles, être imprécises et/ou être incomplètes. Remarquons que la description d'une ressource r_k, qui va être utilisée par le processus w, n'est utile que si la prédiction concernant l'utilisation de cette même ressource par les autres processus est disponible. En conséquence, nous pouvons établir la classification suivante des types d'indicateurs [10] :

1. les *indicateurs mono-dimension et mono-critère*, qui n'expriment la charge qu'au travers d'une unique valeur, ne correspondant qu'à une unique ressource ;

2. les *indicateurs mono-dimension et multi-critères*, qui expriment la charge au travers d'une unique valeur, correspondant à une combinaison de multiples ressources ;

3. les *indicateurs multi-dimensions et multi-critères*, qui expriment la charge au travers de plusieurs valeurs, correspondant à plusieurs ressources.

10. Cette classification est une version simplifiée de la classification proposée dans [Wang 85].

2.6.1 Indicateurs de charge mono-dimension

Un grand nombre de solution proposées utilisent un indicateur basé sur la file d'attente du processeur [Wang 85, Barak 85, Ferrari 88, Efe 89].

L'indicateur proposé par [Barak 85], par exemple, est construit à partir de la moyenne simple de la longueur de cette file. De plus, il tient compte du sur-coût lié au fonctionnement du système opératoire. En notant ω le nombre de quantum où le processeur n'est pas disponible pour les processus utilisateurs, l'indicateur V_t qu'ils proposent est calculé de la façon suivante :

$$V_t = Moy(W^{CPU}, q, T, t) \cdot \frac{\mu}{\mu - \omega} \quad (2.9)$$

$$= \frac{1}{\mu - \omega} \cdot \sum_{i=1}^{\mu} W_i^{CPU} \quad (2.10)$$

Les valeurs qu'ils ont retenues pour mettre en œuvre cet indicateur dans leur système réparti (MOS) sont : $T = 1s$ et $q = 20ms$ (et donc $\mu = 50$).

A l'opposé, les indicateurs de charge moyenne sur 1, 5 et 15 minutes proposés dans les systèmes UNIX, sont calculés à partir de la moyenne amortie exponentiellement du nombre de processus actifs ou endormis depuis moins de 20 secondes (que nous notons $W^{sleep<20s}$). La valeur de ces indicateurs est calculée à partir d'échantillons collectés toutes les cinq secondes ($q = 5s$), représentant respectivement $\mu_1 = 12$, $\mu_5 = 60$ et $\mu_{15} = 180$ échantillons ($\lambda = 1/\mu$)[11]. En prenant q pour unité de temps, l'indicateur UNIX de charge moyenne sur une minute est donc calculé par $load_1 = \widetilde{Moy}(W^{sleep<20s}, 1, \frac{1}{12}, t)$.

De nombreux autres indicateurs ont été considérés dans la littérature. Par exemple, dans [Ferrari 88], plusieurs indicateurs sont proposés et leur performance comparée. La méthode employée compare les performances obtenues par un algorithme de répartition classique utilisant chacun des indicateurs, lorsque les machines sont soumises à des niveaux de charge variables. En plus d'indicateurs simples tels que $load_1$, $M_1 = Moy(W^{CPU}, q_0, T, t)$ (avec $q_0 = 10ms$ et $T = 1s$), ou le taux d'occupation du processeur, les auteurs étudient des indicateurs plus complexes, tel que $\widetilde{M} = \widetilde{Moy}(M_1, q_1, \lambda, t)$ (avec $q_1 = 1s$ et $\lambda \in \{\frac{1}{4}, \frac{1}{20}, \frac{1}{60}\}$[12]), ou des moyennes amorties combinant les longueurs de plusieurs files d'attente (processeur, mémoire, entrées/sorties). Les conclusions de cette étude sont, que d'une façon générale, tous les indicateurs à l'exception de $load_1$ dans un cas de charge, permettent d'obtenir une amélioration du temps de réponse moyen, les meilleures performances étant atteintes avec les indicateurs de type $\widetilde{M}$ calculés sur une période $T = 4s$.

---

11. Les indicateurs proposés par le système Digital Unix (version 4.0) sont configurés différemment : ils utilisent des valeurs échantillonnées toutes les secondes et calculent la moyenne sur 5, 30 et 60 secondes.

12. Notons qu'une erreur s'étant glissée dans la présentation donnée par les auteurs de leur méthode de calcul de la moyenne amortie exponentiellement, nous ne sommes pas absolument certains des valeurs du paramètre λ que nous indiquons.

[Kunz 91] mène une étude comparable à la précédente, mais comparant de nouveaux indicateurs, tels que le nombre de pages mémoire libres, le nombre de changements de contexte, et des combinaisons linéaires ou logiques de plusieurs indicateurs. Les résultats de cette étude, qui ne considère que des indicateurs reflétant des valeurs instantanées, indiquent là aussi que tous les indicateurs permettent d'améliorer les performances par rapport à une exécution sans répartition de charge. Sans surprise, les meilleures performances sont obtenues avec l'indicateur reflétant la taille instantanée de la file d'attente des processus prêts, W^{CPU}.

A côté de ces techniques passives de collecte d'information, signalons aussi l'existence de techniques actives, reposant sur des processus spécialisés. Dans Sprite [Douglis 91], un processus s'exécutant sur chaque machine, avec une faible priorité, incrémente en permanence un compteur. La rapidité avec laquelle le compteur s'incrémente reflète la charge du processeur. Cette technique intrusive présente toutefois plusieurs inconvénients : tout d'abord, en dépit de sa faible priorité, ce processus consomme des ressources qui peuvent faire défaut à d'autres processus. Ensuite, il ne mesure que l'activité de la machine au niveau de l'utilisation du processeur, mais ne permet pas de juger de la charge au niveau d'autres ressources, comme les entrées/sorties. Cependant, cette technique peut parfois s'avérer intéressante. [Richard 98], par exemple, l'utilisent afin de détecter des variations de charge sur des periodes très courtes ($200 \mu s$), sans avoir à instrumenter le noyau du système UNIX. En effet, la résolution des commandes UNIX qui permettent d'observer l'activité du processeur (`vmstat` et `iostat`) est au mieux d'une seconde.

2.6.2 Indicateurs de charge multi-dimensions

Avec l'apparition des applications réparties sur les réseaux de stations de travail, et en particulier des applications parallèles, le problème de répartition dynamique de charge est devenu plus complexe. En effet, les processus de ces applications peuvent être amenés à communiquer entre eux de façon intensive. Or, les communications entre processus placés sur un même site sont beaucoup moins coûteuses que les communications entre processus placés sur des sites différents. Il est donc devenu indispensable de prendre en compte les communications dans la répartition de charge. Cette prise en compte est toutefois délicate, car ses conséquences sur le modèle présenté au paragraphe 2.4 sont importantes. En effet, le temps d'exécution d'un processus ne dépend plus seulement des ressources utilisées par le processus et de la charge d'arrière plan, mais il dépend aussi des processus qui sont présents sur chaque site d'exécution. Par ailleurs, ces applications peuvent être très gourmandes en ressources mémoire, ou générer un volume important d'entrées/sorties. De plus, les entrées/sorties peuvent elles-même générer des communications, car les fichiers peuvent être placés sur des machines distantes.

Par conséquent, pour prendre en compte chacun de ces paramètres et, surtout, leur accorder une importance différente, un indicateur mono-dimension, même s'il est multi-critères, n'est plus suffisant. Certains travaux récents se sont donc intéressés au problème de la définition d'indicateurs multi-dimensions et multi-critères [Folliot 92, Hémery 94, Bernon 95, Chatonnay 98].

[Folliot 92] considère quatre critères :

1. la charge du processeur ;
2. la charge mémoire ;
3. la charge liée à la localisation des fichiers ;
4. la charge liée aux communications.

Ces critères sont organisés de façon hiérarchique par le programmeur de l'application (les informations concernant les ressources utilisées par les processus sont donc connues a priori). Cette organisation hiérarchique permet d'appliquer un algorithme d'affinages successifs de la sélection : un ensemble de machine est retenu selon le premier critère ; si cet ensemble comporte plus d'une machine, l'algorithme considère alors le second critère et abouti à un second ensemble, et ainsi de suite. La charge liée aux communications est traitée en considérant, d'une part, le nombre de processus communiquant et, d'autre part, le volume de leurs communications. Comme le mécanisme de répartition a tendance à placer les tâches sur un même site (phénomène d'attraction), la possibilité est donnée au programmeur du système (Gatos), d'interdire à deux tâches d'être placées sur un même site (phénomène de répulsion).

[Hémery 94] aborde le problème de façon différente, en combinant à la fois une analyse statique du comportement des applications et une optimisation de sa politique de placement par l'analyse hors ligne de traces d'exécution. À cela, s'ajoutent plusieurs politiques de répartition dynamique de charge. Ces politiques sont proposées au programmeur au travers du paradigme objet. C'est donc le programmeur qui construit la politique de répartition dynamique, en choisissant notamment de quelle façon utiliser les indicateurs proposés par l'environnement d'exécution. Ces indicateurs sont au nombre de trois :

1. le nombre de processus présents sur un site ;
2. le nombre de processus dans l'état prêt sur un site ;
3. le nombre de messages en attente de traitement.

Dans [Bernon 95], les phénomènes d'attraction/répulsion introduit par [Folliot 92] ne s'appliquent plus entre paires de processus, mais entre processus et site d'exécution. La répulsion est donc vue comme une attirance moins forte d'un processus pour un site. L'attirance d'un processus pour un site est calculée à partir des critères suivants :

1. les processus du site qui communiquent avec le processus pour lequel l'attirance est calculée ;
2. la capacité de traitement du site (qui est fonction de la charge processeur) ;
3. la capacité mémoire du site ;

4. les contraintes concernant les ressources dites classiques, comme les fichiers et les périphériques.

Le pouvoir d'attraction d'un site est réévalué à chaque allocation ou libération d'une ressource.

[Chatonnay 98] étudie la répartition dynamique de charge dans le cadre d'un système réparti à objets s'appuyant sur CORBA [The OMG Consortium 93]. Les critères utilisés sont :

1. la charge du processeur ;
2. les relations de dépendance entre objets (qui génèrent des communications) ;
3. les contraintes de positionnement relatives aux composants de l'application ;

Chaque critère est défini sur un intervalle de valeurs borné : plus le critère a une valeur élevée, plus l'action qu'il évalue est intéressante. Un droit de veto est associé à chacun des critères, afin d'interdire l'action évaluée. Un critère de synthèse est formé à partir d'une combinaison linéaire des trois critères précédents. La somme des coefficients de cette combinaison est normalisée. Le coefficient associé au critère de dépendance est ajusté à partir de l'analyse dynamique de l'intensité des relations entre objets.

2.7 Prise en compte de l'hétérogénéité

L'hétérogénéité, nous l'avons dit, est une caractéristique importante des réseaux de stations de travail, car elle apparaît à tous les niveaux. L'hétérogénéité des architectures est généralement prise en compte en multipliant la valeur de la charge mesurée par un facteur proportionnel à la puissance des processeurs [Barak 85, Folliot 92, Chatonnay 98] : si $\sigma(X)$ représente la puissance du processeur X et $\sigma(Y)$ la puissance du processeur Y, et que $\sigma(X) = k \cdot \sigma(Y)$, alors la charge du processeur X est divisée d'un facteur k.

Notons toutefois que ce type de correction n'est pas parfaite car la différence des performances entre processeurs n'est pas homogène selon les opérations. Par exemple, certaines architectures super-scalaires comportent plusieurs unités de traitement flottantes ou arithmétiques alors que d'autres n'en comportent qu'une. Les performances de ces unités de traitement, ensuite, sont très variables, selon la richesse des jeux d'instruction, la taille des pipelines ou la taille des registres (de 32 à 128 bits) [Seznec 96, Seznec 97].

Mais nous l'avons vu, le processeur n'est pas la seule source d'hétérogénéité. Celle-ci apparaît aussi au niveau des systèmes d'exploitation, des réseaux, des entrées/sorties ou des configurations. Certains travaux, en particulier [Folliot 92], prennent en compte certaines de ces formes d'hétérogénéité, en appliquant le principe du facteur de puissance à plusieurs ressources. Les solutions proposées sont généralement manuelles, c'est-à-dire que le calcul des coefficients doit être pris en charge par un utilisateur ou par l'administrateur de la configuration. Ces solutions sont donc difficilement gérables lorsque le nombre des paramètres à évaluer est important et change fréquemment.

2.8 Conclusion

Dans ce chapitre, nous avons présenté la problématique de la construction d'indicateurs de charge dans les architectures réparties faiblement couplées. Pour cela, nous avons d'abord présenté les caractéristiques de ces plate-formes qui ont une influence sur ce problème. Nous avons ensuite présenté le problème de la répartition dynamique de charge, et les objectifs recherchés. Puis nous avons décrit les deux facettes du problème de la répartition dynamique de charge : d'un côté, les informations concernant la description des ressources requises par les processus qui doivent être placés, et de l'autre les indicateurs de charge, qui permettent de choisir la machine qui correspond le mieux aux ressources demandées.

Ensuite, nous avons décrit les indicateurs de charge les plus utilisés dans la littérature et les méthodes permettant de les calculer. Pour cela, nous avons distingué deux grandes classes d'indicateurs : les indicateurs mono-dimension, qui reflètent l'activité d'une machine au travers d'une unique quantité. Nous avons ensuite décrit les travaux les plus récents sur ce sujet, dans lesquels des indicateurs multi-dimensions sont proposés. Ces indicateurs sont particulièrement adaptés dans le contexte d'utilisation actuel des réseaux de stations de travail. En effet, l'évolution des performances des composants utilisés dans les réseaux de stations de travail, rend ces derniers de plus en plus compétitifs face aux autres architectures réparties (en particulier les machines parallèles). Les principales applications clientes de la répartition dynamique de charge dans ce contexte sont dorénavant les applications parallèles et réparties. Les ressources consommées par ces dernières ne sont plus exclusivement orientées calcul, mais accordent une part importante aux communications, à la mémoire, et aux entrées/sorties.

Ce type de comportement oblige donc à considérer des indicateurs de charge multi-dimensions et multi-critères. Toutefois, la difficulté de construction des indicateurs dans ce nouveau contexte ne se limite pas au problème de la multiplication des ressources significatives. En effet, nous posons aussi le problème de la forte hétérogénéité des réseaux de stations de travail, qui ajoute encore à cette complexité, à tel point que les méthodes manuelles de construction d'indicateurs de charge deviennent pratiquement inapplicables.

Par conséquent, nous en concluons qu'il devient indispensable de trouver des méthodes pour automatiser la construction de ces indicateurs. [Mehra 93] aborde le problème en proposant une méthode basée sur des techniques d'apprentissage (réseaux de neurones). Toutefois, la méthode proposée vise la répartition de charge par migration, technique qui ne supporte qu'une faible hétérogénéité de l'architecture. Le problème reste donc ouvert dans le cas d'architectures fortement hétérogènes.

Au chapitre 4, nous étudions ce problème et proposons une méthodologie pour faciliter la construction automatique d'indicateurs de charge multi-critères en environnement fortement hétérogène.

Chapitre 3

Communications multipoints

3.1 Introduction

Les communications multipoints, par opposition aux communications point-à-point, sont des communications qui font intervenir plus de deux interlocuteurs. Ce type de communications, et en particulier la diffusion (*broadcast*, *multicast*), qui consiste à transmettre une même information à plusieurs destinataires, est utilisée (ou potentiellement utilisable) par un nombre sans cesse grandissant d'applications.

Comme le soulignent [Mankin 98], il existe une grande diversité des besoins en matières de communications multipoints : certaines applications exigent un ordre total sur la transmission des messages alors que d'autres n'en n'ont pas besoin. Dans certaines applications, tous les membres envoient des données alors que dans d'autres il n'existe qu'un seule source. Dans certaines applications, les données ne sont présentes qu'en un seul endroit, alors que dans d'autres elles sont répliquées, ce qui autorise plusieurs membres à les transmettre ou retransmettre. Certaines applications ont un nombre statique et petit de membres, d'autres doivent pouvoir s'étendre dynamiquement à des milliers, voire des dizaines de milliers de membres. Certaines applications ont des besoins importants et constants de bande passante, d'autres ont des besoins ponctuels imprévisibles n'exigeant pas beaucoup de bande passante, mais des temps de réponse réduits. Certaines applications ont des contraintes temps réel importantes que d'autres n'ont pas. Certaines applications peuvent tolérer de sacrifier en partie la fiabilité des transmissions si cela leur permet d'améliorer de façon sensible les temps de transmission.

Les communications multipoints sont étudiées au sein de quatre communautés de recherche, ayant des thématiques relativement différentes, bien que convergentes sur certains points : la communauté Internet, qui s'intéresse aux aspect purement réseaux et protocoles, la communauté des systèmes d'exploitation, qui s'intéresse aux système d'exploitation répartis, une partie de la communauté de l'algorithmique répartie, qui s'intéresse aux problèmes de tolérance aux pannes, et la communauté du parallélisme, qui s'intéresse au environnements de programmation et au support d'exécution d'applications parallèles.

Avant de décrire les réponses apportées au sein de chacune de ces communautés, dans les paragraphes 3.3 à 3.5, nous allons d'abord décrire, les principales caractéristiques de ces communications.

3.2 Caractéristiques des communications multipoints

Les différents mécanismes de communication multipoints se distinguent en premier lieu par les aspects sémantiques des communications qu'ils proposent, ainsi que par la façon dont ils gèrent les ensembles de processus impliqués dans les communications ; les groupes de communications, que nous présentons au paragraphe 3.2.1 sont en général utilisés pour définir ces caractéristiques. En second lieu, ces mécanismes se distinguent aussi par les modes de transmission qu'ils proposent, ce dont nous discutons au paragraphe 3.2.2. Enfin, ils se distinguent aussi par les schémas de communication qu'ils offrent, et que nous décrivons au paragraphe 3.2.3.

3.2.1 Groupes de communication

Un groupe de communication est une abstraction qui permet de désigner collectivement un ensemble de processus. Dans les paragraphes qui suivent, nous présentons leurs caractéristiques principales.

Dynamicité. Un groupe est *dynamique* lorsqu'il permet aux processus de se joindre au groupe ou de le quitter à tout moment. On peut distinguer deux types de groupes dynamiques : les groupes *dynamiques persistants* et les groupes *dynamiques transitoires*. Un groupe dynamique persistant est un groupe qui survit à la disparition de son dernier membre, alors qu'un groupe dynamique transitoire est détruit lorsque son dernier membre le quitte.

Modalités d'accès. Les modalités d'accès à un groupe de communication indiquent dans quelle mesure les processus n'appartenant pas un groupe de communication ont la permission d'accéder à ses fonctionnalités. [Tanenbaum 95], par exemple, ne considère cette accessibilité qu'au niveau de l'émission des messages : il appelle *groupe ouvert*, un groupe dans lequel des processus n'appartenant pas au groupe ont le droit d'envoyer des messages, et *groupe fermé*, un groupe dans lequel seuls les processus membres ont le droit d'envoyer des messages.

Égalité des processus. Les groupes sont dit *égalitaires* lorsqu'aucun des processus membres du groupe n'y joue un rôle particulier. Dans le cas contraire, les groupes sont habituellement qualifiés de *hiérarchiques*.

Gestion répartie. On dit que la gestion d'un groupe est *répartie* lorsque tous les nœuds jouent le même rôle dans la gestion d'un groupe. Elle est *centralisée* lorsqu'un unique nœud doit assumer le rôle de coordinateur. Elle est *hiérarchique* lorsque la coordination est prise en charge par de multiples nœud, chacun responsable d'un sous-ensemble des nœuds.

Identifiants de groupe. L'identifiant du groupe est *logique* lorsqu'il peut être choisi indépendamment des protocoles de communication sous-jacents et *physique* dans le cas contraire. Par ailleurs, l'identifiant est *explicite* lorsque chacun des destinataires doit être désigné individuellement et *implicite* lorsqu'ils peuvent être désignés collectivement.

Séquencement. On distingue généralement les types de séquencement suivants :

1. *aucun séquencement* : les messages sont envoyés et/ou reçus de façon désordonnée ;
2. *séquencement* FIFO *selon la source* : les messages sont reçus à destination dans l'ordre où ils ont été envoyés par leur expéditeur ;
3. *séquencement causal* : comme précédemment, les messages sont reçus dans l'ordre où ils ont été envoyés. De plus, si un message M1 est diffusé et que ce message entraîne la diffusion d'un message M2 par l'un des destinataires de M1, alors le message M1 doit être reçu avant le message M2 par tous les destinataires des deux messages (autrement dit, on ne peut recevoir la réponse à un message avant d'avoir reçu le message initial) ;

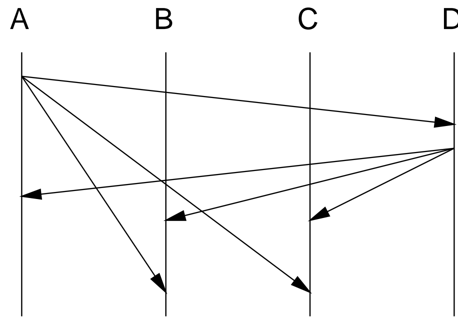


FIG. 3.1: *Exemple d'un séquencement atomique non causal*

4. *séquencement atomique* : tous les destinataires reçoivent tous les messages dans le même ordre. Mais cet ordre ne respecte pas nécessairement l'ordre exact des émissions. Comme le montre la figure 3.1, cette propriété est orthogonale avec la précédente. Dans cet exemple, le nœud A diffuse un message vers les nœuds B, C et D. En réception de ce message, D diffuse à son tour un message vers les nœuds A, B et C, mais celui-ci est délivré sur les nœuds B et C avant le message diffusé par A. La contrainte de séquencement atomique est respectée car les deux seuls nœuds à recevoir les deux messages, B et C, les reçoivent dans le même ordre. En revanche la contrainte de causalité n'est pas respectée, car B et C reçoivent la "réponse" de D au message de A avant le message de A ;

5. *séquencement atomique causal* : les propriétés des deux séquencements précédents sont respectées ;

6. *séquencement synchrone* : les messages sont reçus dans l'ordre respectif de leur date d'émission par rapport à une horloge globale. Précisons que dans une architecture répartie faiblement couplée, cette horloge globale ne peut être que virtuelle, car les nœuds de l'architecture ne disposent pas d'une horloge physique commune ;

Lorsqu'un processus appartient simultanément à plusieurs groupes, on distingue aussi le cas où les propriétés du séquencement sont assurées de façon globale pour tous les groupes (ce que l'on appelle un "séquencement inter-groupe consistant") de celui où ces propriétés ne sont assurées que sur la base du groupe seul.

Multiplexage. Le multiplexage permet de distinguer plusieurs flots de communication au sein d'un groupe de communication. Cette fonctionnalité, qui n'est pas toujours présente, permet d'éviter les interférences entre composants logiciels d'une même application (bibliothèques) ou entre applications concurrentes. Elle permet aussi d'attribuer des niveaux de priorité aux messages.

Le multiplexage s'obtient par l'association d'attributs à chaque message transmis. Un attribut peut être une simple étiquette (un entier, par exemple) ou représenter un prédicat que doit vérifier chaque destinataire : prédicat sur le numéro ou l'état du processus, prédicat sur l'état de charge du site, etc.

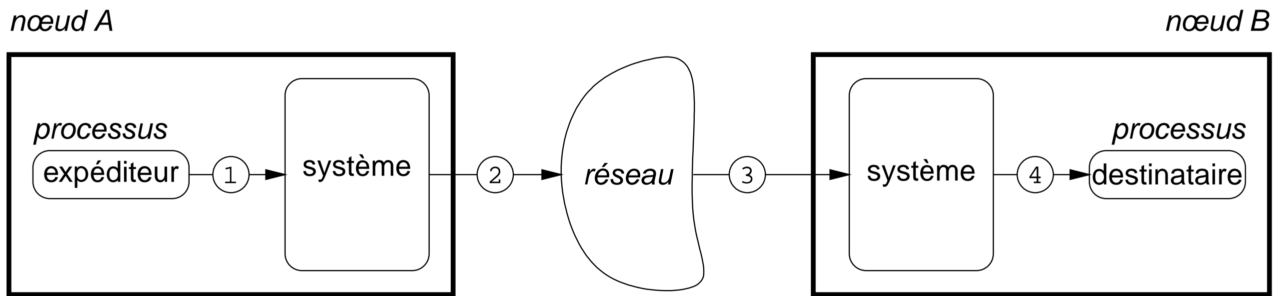


FIG. 3.2: *Étapes d'une transmission de message*

Atomicité. La diffusion d'un message est dite *atomique* lorsque, soit tous les destinataires reçoivent le message, soit aucun d'entre eux ne le reçoit.

3.2.2 Modes de transmission

La réalisation d'une communication s'effectue en quatre étapes (figure 3.2) :

① Le message est confié par le processus expéditeur au système chargé de la transmission (système d'exploitation ou environnement de programmation) ;

② Le système de l'expéditeur place le message sur le réseau ;

③ Le système du destinataire récupère le message depuis le réseau ;

④ Le message est délivré par le système au processus destinataire.

Les modes de transmission définissent la sémantique des opérations d'émission et de réception par rapport au franchissement de ces étapes.

- une opération d'émission est *synchrone* lorsqu'elle permet à l'expéditeur de s'assurer que son message a bien été reçu, c'est-à-dire qu'il a franchi la troisième étape (synchronisation faible) ou la quatrième étape (synchronisation forte) ;

- une opération d'émission est *asynchrone* lorsqu'elle ne permet pas toujours à l'expéditeur de s'assurer que son message a bien été reçu ;

- une opération d'émission est *bloquante* lorsqu'elle autorise le système à bloquer (endormir) le processus expéditeur durant tout ou partie de la transmission ;

- une opération d'émission est *non bloquante* lorsqu'elle interdit au système de bloquer le processus expéditeur ;

- une opération de réception est *bloquante* lorsqu'elle exige du système qu'il bloque le processus appelant lors de la quatrième étape jusqu'à ce qu'un message puisse être délivré ;

– une opération de réception est *non bloquante* lorsqu'elle ne bloque pas le processus appelant quand aucun message n'est disponible ;

– une opération (émission ou réception) est *atomique* lorsqu'elle peut être complètement réalisée par un unique appel de fonction ;

– une opération est *non atomique* lorsqu'elle requiert plusieurs appels de fonction. Par exemple, une opération d'émission synchrone et non bloquante exige en principe plusieurs appels de fonction ;

3.2.3 Schémas de communication

On peut définir trois grandes catégories de schémas de communication multipoints :

– les communications de type "un-vers-plusieurs" (*one-to-many*), pour lesquelles un unique expéditeur envoie des données vers plusieurs destinataires ;

– les communications de type "plusieurs-vers-un" (*many-to-one*), pour lesquelles de multiples expéditeurs envoient des données vers un unique destinataire ;

– les communications de type "plusieurs-vers-plusieurs" (*many-to-many*), pour lesquelles de multiples expéditeurs envoient des données vers plusieurs destinataires.

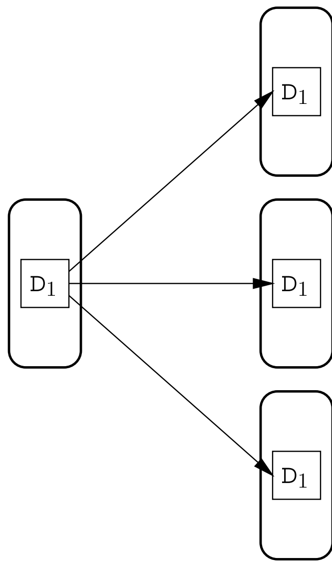
3.2.3.1 Schémas de type "un-vers-plusieurs"

On distingue à nouveau deux catégories de schémas de communication dans cette classe : les schémas de *diffusion*, dans lesquels l'expéditeur envoie le même message vers chacun des destinataires, et les schémas de *distribution*, dans lesquels il envoie des messages différents à chacun des destinataires. Chacun de ces deux schémas se décompose à son tour en deux sous-catégories : les diffusions et distributions *partielles*, dans lesquelles les messages ne sont envoyés que vers un sous-ensemble des destinataires possibles (selon le contexte, il peut s'agir des membres d'un groupe de communication ou des nœuds d'une architecture répartie), et les diffusions et distributions *généralisées*, dans lesquelles les messages sont envoyés vers la totalité des destinataires possibles. La figure 3.3 représente chacun de ces quatre schémas de communication.

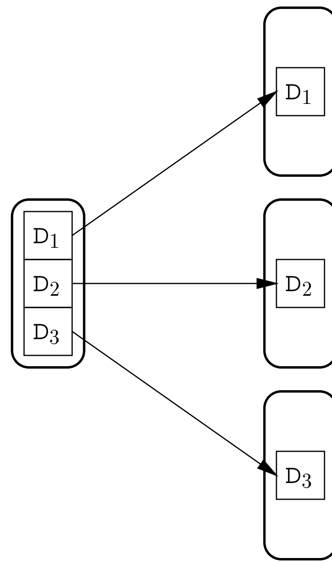
Les schémas de distribution sont surtout utilisés par les applications parallèles [Bertsekas 89]. Les schémas de diffusion, en revanche, sont très largement utilisés par l'ensemble des applications réparties.

Par ailleurs, précisons que la terminologie purement francophone que nous avons employé n'est pas très répandue. On lui préfère généralement une terminologie anglicisée. Ainsi, la diffusion généralisée est simplement appelée diffusion (ou *[single node]broadcast[ing]*), la diffusion partielle est appelée *multicast[ing]* et les deux formes de distribution sont appelées *[single node]scatter[ing]*[1]. Par la suite, nous utiliserons la terminologie suivante, qui est la plus couramment utilisée : diffusion, *multicast* et *scater*.

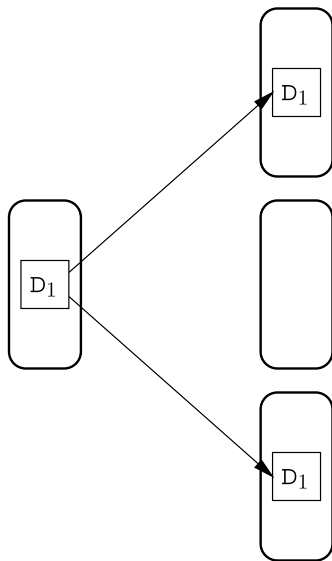
1. Les deux formes de distribution ne sont que rarement différenciées car d'une part, elles sont très spécialisées, et d'autre part, à l'inverse des deux formes de diffusion, leur niveau de complexité est très comparable.



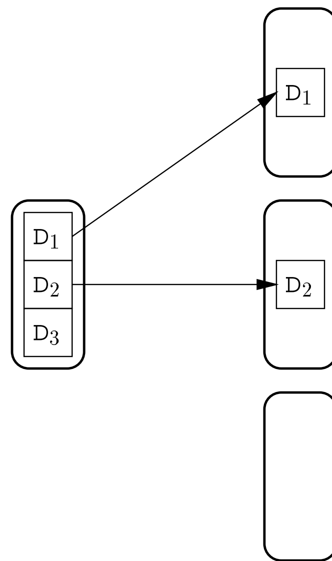
(a) Diffusion généralisée



(b) Distribution généralisée



(c) Diffusion partielle



(d) Distribution partielle

FIG. 3.3: *Schémas de communication de type "un-vers-plusieurs"*

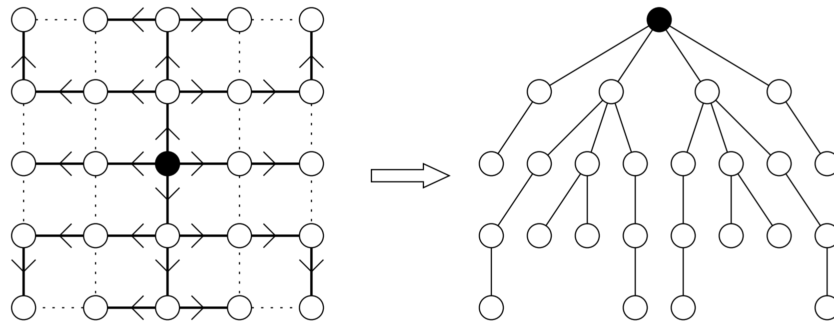


FIG. 3.4: *Exemple d'arbre de diffusion sur une grille*

La topologie d'une architecture répartie est généralement représentée par un graphe, dont les sommets sont les nœuds de l'architecture et les arcs (ou arêtes) sont les liens du réseau qui relient entre eux chacun de ces nœuds [Rumeur 94]. Un *arbre de diffusion* est un arbre recouvrant de ce graphe (*spanning tree*), dont la racine est l'expéditeur du message et dont les nœuds et feuilles sont les destinataires. La figure 3.4 présente un exemple d'arbre de diffusion construit à partir du centre d'une grille de 5×5 nœuds.

La mise en œuvre efficace des schémas de communication que nous venons de voir exige, en général, de connaître (ou d'apprendre) la topologie du réseau, afin de construire un ou plusieurs arbres de diffusion. L'utilisation d'un arbre de diffusion permet de réduire le nombre de messages qu'il faut envoyer : la racine ne transmet le message que vers ses fils directs, qui les diffusent de la même façon dans le sous-arbre dont ils sont racine (c'est-à-dire uniquement vers leurs fils directs, qui les diffusent à leur tour vers leurs propres fils, et ainsi de suite). Autrement dit, chaque message ne circule au plus qu'une seule fois sur chaque lien du réseau.

3.2.3.2 Schémas de type "plusieurs-vers-un"

D'un point de vue purement communication, il n'existe qu'un seul schéma de communication de ce type. Il s'agit du schéma dual de la distribution, dans lequel chacun des nœuds envoie un message vers un même destinataire. On l'utilise, par exemple, à la suite d'une diffusion pour permettre à chacun des nœuds d'accuser réception du message diffusé. Pour mettre en œuvre de façon efficace ce type de communication, on réutilise généralement les arbres de diffusions précédents, mais en sens inverse, des feuilles vers la racine. La technique la plus simple consiste à faire remonter systématiquement chacun des messages le long de l'arbre, vers la racine, séparément ou en les agrégeant. On parle dans ce cas de rassemblement (*[single node]gather[ing]*).

Toutefois, cette technique conduit rapidement à un volume de données important sur les liens de communication, au fur et à mesure que l'on s'approche de la racine. De plus, le traitement de très nombreux messages par la racine peut se révéler très coûteux. Une technique couramment utilisée pour résoudre ces problèmes consiste à essayer de combiner, sur chaque nœud de l'arbre, les messages reçus des fils en un seul message de taille comparable à celle des messages reçus. Cette combinaison s'obtient généralement par l'application d'un opéra-

teur binaire associatif. On parle dans ce cas de *réduction*, de *combinaison* ou d'*accumulation* (*reduce[ing]*, *[single node]accumulation*)

Par exemple, dans le cas de l'acquittement d'un message diffusé, chaque nœud de l'arbre n'envoie qu'un seul message d'acquittement vers son père, pour la totalité du sous-arbre dont il est racine : si le message a bien été reçu par tous les nœuds de ce sous-arbre, l'acquittement est positif, dans le cas contraire il est négatif. Ainsi la racine ne reçoit plus qu'un acquittement de la part de chacun de ses fils directs, plutôt qu'un de la part de chacun des nœuds et feuilles de l'arbre.

3.2.3.3 Schémas de type "plusieurs-vers-plusieurs"

Les schémas de ce type sont des combinaisons des schémas élémentaires précédents. Ils sont donc trop nombreux pour que l'on puisse les distinguer tous. Les plus étudiés sont ceux dans lesquels la même opération est répétée plusieurs fois :

– la *multi-diffusion* (*multinode broadcast[ing]*) est le schéma dans lequel plusieurs nœuds déclenchent la diffusion d'un message vers les autres nœuds ;

– la *multi-distribution* (*multinode scatter[ing]*) est le schéma dans lequel plusieurs nœuds déclenchent une distribution vers les autres nœuds ;

– le *multi-rassemblement* (*multinode gather[ing]*) est le schéma dans lequel des rassemblements sont déclenchés vers plusieurs nœuds ;

– la *multi-reduction* (*multinode accumulation, multinode reduction*) est le schéma dans lequel des réductions sont déclenchées vers plusieurs nœuds ;

Une multi-diffusion déclenchée à partir de l'ensemble des nœuds est aussi appelée *gossip[ing]*. L'échange total (*total exchange*) est le schéma dans lequel chaque nœud envoi un message différent vers chacun des autres nœuds. On peut donc le considérer indifféremment comme une multi-distribution lancée depuis chaque nœud ou comme un multi-rassemblement ou une multi-réduction déclenchée vers chaque nœud.

3.3 Protocoles

Les protocoles sont l'indispensable fondement de tout mécanisme de communication. Nombre d'entre eux ont été proposés, et continuent d'être proposés, pour tenter de répondre au difficile problème des communications multipoints fiables. Mais selon la nature des sous-problèmes sur lesquels ils se focalisent, ces protocoles aboutissent souvent à des solutions très différentes. Dans les paragraphes qui suivent, nous illustrons cette diversité en présentant une sélection représentative de ces protocoles. Nous commençons par IP/Multicast, car il fournit le mécanisme de diffusion élémentaire sur lequel reposent la majorité des protocoles présentés. Nous présentons ensuite les protocoles MTP, XTP, RAMP et RMTP, qui sont représentatifs des principales solutions proposées pour mettre en œuvre des groupes de communication dynamiques offrant un minimum de séquencement.

3.3.1 IP/Multicast

IP/Multicast [Deering 89, Huitema 95] est un protocole expérimental de *multicast*, prévu pour fonctionner au niveau transport, c'est-à-dire entre des machines qui ne sont pas nécessairement placées sur un même réseau local. Ce protocole s'utilise soit directement au-dessus du protocole IP, soit au-dessus du protocole UDP. Il propose la même qualité de service que le protocole au-dessus duquel il est utilisé (IP ou UDP). Il s'agit donc d'un protocole de type *best-effort*, qui ne garantit ni la fiabilité, ni le séquencement des transmissions.

Ce protocole est expérimental, mais son implantation, le MBONE (*Multicast backbone*), progresse régulièrement, notamment en raison du succès des applications multimédia qu'il permet de supporter (visio-conférence, tableaux blancs, etc). Par ailleurs, ce protocole a été officiellement incorporé au standard IPv6 [Huitema 96], ce qui, à long terme, en garantit la généralisation. Les mécanismes mis en œuvre par ce protocole peuvent être regroupés en deux catégories : les mécanismes intra-réseau et les mécanismes inter-réseau.

Au niveau intra-réseau, ces mécanismes prennent en charge, d'une part, l'enregistrement des machines d'un réseau local dans un groupe de *multicast* (grâce au protocole IGMP [Deering 89]), et d'autre part, l'acheminement des messages multipoints vers leurs destinataires sur le réseau local. Les identifiants de groupe sont physiques : ce sont les adresses IP de classe “D” (adresses IP dont les quatre bits de poids fort sont 1110).

Au niveau inter-réseau, ces mécanismes cherchent à assurer un routage efficace des messages multipoints émis dans un groupe. En fait, les recherches menées autour de ce protocole se sont surtout intéressées à l'optimisation de cette deuxième catégorie de mécanismes. Ce protocole n'apporte donc pas de réponse originale au problème du *multicast* à l'échelle d'un réseau local. Il se contente de réutiliser la capacité de *multicast* au niveau liaison des réseaux locaux, quand elle est disponible.

À ce titre, une partie des adresses de liaison affectées au *multicast* par la norme IEEE 802 sont réservées au protocole IP/Multicast. Ces adresses sont de la forme `01-00-5E-xx-xx-xx` (en notation héxadécimale). Les 23 bits de poids faible de l'adresse IP désignant le groupe destinataire sont placés dans les 23 bits de poids faible de l'adresse de destination de niveau liaison [Huitema 95].

IP/Multicast n'apporte donc pas de réponse directe au problème des communications multipoints fiables et ordonnées. En revanche, il fournit les mécanismes de base minimaux pour la construction de tels protocoles, tant sur Internet que sur la majorité des réseaux locaux.

3.3.2 MTP

MTP [Armstrong 92] est un protocole de niveau transport conçu pour fournir des communications multipoints fiables et efficaces au-dessus des protocoles de niveau réseau existants, tel que IP/Multicast. Ce protocole permet la création de groupes de communication, qui sont appelés des *toiles* (*web*). Les membres d'une toile peuvent être de trois types : les consommateurs seuls, les producteurs/consommateurs et le maître, qui est aussi producteur

et consommateur. Il n'existe qu'un seul maître par toile. Ce maître contrôle tous les aspects des communications dans la toile.

Le maître doit initialiser la toile avant que celle-ci ne soit opérationnelle (il doit donc être lancé le premier). Cette initialisation consiste à s'assurer que l'adresse de niveau réseau qui va être utilisée pour réaliser les communications dans la toile n'est pas déjà affectée. Lorsque l'initialisation est achevée, les membres peuvent demander à s'enregistrer dans la toile auprès du maître. Ils utilisent pour cela un identificateur de connexion unique, dont la génération et la prise en charge est externe au protocole. La requête d'enregistrement comporte plusieurs paramètres. Ces paramètres indiquent si le demandeur désire être consommateur simple ou producteur/consommateur, si les transmissions doivent être fiables ou *best-effort* et si le demandeur est capable d'accepter plusieurs expéditeurs d'informations (schémas de communication de type "plusieurs-vers-plusieurs"). Ils fixent aussi le débit minimal souhaité et la taille maximale d'un paquet de données. Si la requête peut être satisfaite, le maître répond avec un acquittement positif contenant l'identifiant de la connexion multipoints ; dans le cas contraire un acquittement négatif est retourné.

Le maître contrôle le séquencement et le débit des données envoyées à l'aide d'un mécanisme de jeton. Pour qu'un producteur puisse envoyer des données, il faut qu'il obtienne un jeton du maître. Ce jeton lui permet d'envoyer une certain nombre de paquets, qui définissent la *fenêtre* de transmission. Les membres qui ne sont pas capables de supporter le débit minimal imposé par la toile en sont exclus. Les erreurs de transmission sont gérées à l'aide d'acquittements négatifs. Ces acquittements sont envoyés au producteur lorsque l'émission de sa fenêtre se termine. Ces demandes de retransmission sont sélectives : seuls les paquets perdus sont retransmis. Les paquets manquants sont rediffusés dans la toile. Ce mécanisme implique donc que les producteurs doivent conserver les données transmises un temps suffisamment long, et que les consommateurs doivent être capable de détecter les transmission redondantes.

MTP présente certes des qualités, comme le contrôle du débit par les membres du groupe, des acquittements négatifs, la retransmission sélective et, surtout, les schémas de communication de type "plusieurs-vers-plusieurs". Mais l'approche centralisée autour d'un unique maître introduit automatiquement un point de congestion au niveau de ce dernier, ce qui nuit à l'extensibilité du protocole. Par ailleurs, la nécessité pour les producteurs de demander puis d'attendre un jeton introduit un délai systématique dans les transmissions. Enfin, comme ce protocole n'intègre aucun mécanisme de contrôle de congestion, et qu'il s'impose des contraintes temps réel, il ne supporte que très mal les variations importantes de la bande passante disponible. Toutes ces caractéristiques le destinent donc plutôt à une utilisation sur un réseau local, en contexte temps réel, plutôt que dans un contexte haute performance.

3.3.3 XTP

XTP 4.0 (Xpress Transfert Protocol) [XTP Forum 95] est un protocole de niveau transport, qui a été conçu pour répondre aux besoins d'une grande variété d'applications, depuis les applications temps réel embarquées jusqu'aux services de distribution de multimédia et

aux applications réparties à grande échelle. De même que TCP, UDP, IP/Multicast ou TP4, XTP peut être placé au-dessus d'un protocole de niveau réseau tel que IP ou CLNP[2]. Mais il peut aussi être placé directement au-dessus de la couche de liaison (LLC, MAC) ou au-dessus de la couche AAL de ATM. Les qualités de ce protocole sont sa grande richesse fonctionnelle et sa flexibilité.

En effet, ses fonctionnalités lui permettent de proposer des services identiques à ceux des protocoles TCP, UDP, TP4 et IP/Multicast, ainsi que des communications multipoints fiables et ordonnées selon la source. Ces communications s'appuient sur des groupes de communication. Sa flexibilité vient de sa capacité à gérer de façon orthogonale la plupart des propriétés courantes d'un protocole :

- **contrôle d'erreur** : XTP propose un mode de transmission fiable reposant sur des acquittements positifs (comme celui de TCP), un mode fiable reposant sur des acquittements négatifs rapides (plus efficace sur les réseaux locaux), et un mode non fiable (tel que celui de UDP) ;

- **retransmission** : en cas de perte, XTP propose trois stratégies de retransmission. La première consiste simplement à ignorer la perte, comme dans UDP. La seconde est une stratégie de type retour arrière (*go-back-n*), dans laquelle toutes les données non encore acquittées sont renvoyées, même lorsque certaines ont déjà été reçues, comme dans TCP. La troisième est une stratégie de retransmission sélective avec laquelle seules les données manquantes sont retransmises ;

- **contrôle de flux** : XTP propose trois politiques de contrôle de flux. La première consiste simplement à ne placer aucune limitation, comme dans UDP. La seconde s'appuie sur un mécanisme de fenêtrage, comme dans TCP. La troisième est une politique de réservation, dans laquelle l'application définit la quantité de mémoire tampon qui est allouée à chaque connexion ;

- **correction d'erreur** : le calcul des mots de parité sur les données transmises peut être désactivé ;

- **gestion des acquittements** : la fréquence des acquittements est contrôlée par la source ; l'émetteur indique au destinataire quand ce dernier doit lui envoyer un acquittement pour les données reçues ;

- **données hors bande** : une étiquette de 64 bits peut être associée à chaque transmission ;

- **gestion de priorité** : XTP permet l'association d'un niveau de priorité sur 16 bits à chaque message ;

2. TP4 et CLNP sont les équivalents respectifs pour la suite protocole OSI des protocoles TCP et IP dans la suite de protocole TCP/IP.

- **contrôle de débit** : XTP permet à un receveur de contrôler dynamiquement le débit des émissions par l'émetteur (parallèlement au mécanisme de contrôle de flux) au travers de deux paramètres, *burst* et *credit*. Le premier définit la quantité maximale de données qui peut être envoyée en une transmission et le second définit la quantité maximale de données qui peuvent être envoyée par unité de temps ;

- **descripteurs de traffic** : XTP prévoit un champ de description de traffic, qui peut être utilisé pour fixer des paramètres de qualités de service sur les réseaux qui supportent cette fonctionnalité (comme ATM par exemple).

Chacune de ces fonctionnalités est disponible tant avec des communications point-à-point qu'avec des communications multipoints. Les communications multipoints s'appuient sur des groupes de communication, appelés groupes de *multicast*. Les groupes de *multicast* ne sont pas égalitaires : un groupe ne contient qu'un seul membre écrivain et un nombre arbitraire de consommateurs. Lorsque des schémas de communication de type plusieurs-vers-plusieurs sont requis, le protocole exige donc la définition (superposition) de plusieurs groupes de *multicast*. L'utilisateur peut autoriser ou interdire sélectivement l'admission d'un nœud consommateur dans un groupe de communication. Les groupes sont dynamiques et l'application peut à tout moment en obtenir la liste des membres. L'application définit la façon selon laquelle le protocole doit réagir lors de l'apparition ou de la disparition d'un nouveau membre consommateur dans le groupe.

XTP est donc un protocole multipoints très complet. Sa grande richesse fonctionnelle et sa flexibilité lui permettent donc, *potentiellement*, de répondre à tous les types de besoins. Potentiellement, car son architecture concentrique (tous les receveurs sont connectés à la source) limite son extensibilité. De plus, si un schéma de type "un-vers-plusieurs" est suffisant pour construire des schémas de type "plusieurs-vers-plusieurs", cette stratégie n'est pas nécessairement la plus efficace. Tout d'abord, elle exige l'établissement de nombreuses connections XTP et, par suite, la gestion de nombreux flots de communication concurrents. Ensuite, cette stratégie ne permet pas d'assurer un contrôle de flux global entre les flots.

Ajoutons par ailleurs que le nombre important de fonctionnalités qui sont proposées se traduisent, en pratique, par une complexité de mise en œuvre importante.

3.3.4 RAMP

RAMP (*Reliable Adaptive Multicast Protocol*) [Braudes 93] est un protocole de niveau transport conçu pour fournir un service multipoints fiable, en mode connecté, au-dessus de protocoles de niveau réseau tels que IP/Multicast.

Ce protocole s'accompagne d'un service de gestion globale des groupes de communication, le MGA (*Multicast Group Authority*), qui est aussi chargé de l'attribution des identifiants de groupe (les adresses de la classe "D" avec IP/Multicast). Ces groupes de communication sont dynamiques.

Initialement, ce protocole a été élaboré afin de permettre la transmission fiable d'images digitales de très grande taille (plusieurs dizaines de milliers de pixels de côté) vers de multiples utilisateurs. La transmission de ce type de données peut souvent être divisée en de multiples flots, organisés de façon hiérarchique, correspondant à des niveaux de qualité différents. De cette façon, les destinataires qui ne peuvent recevoir les données qu'à faible débit ne reçoivent qu'une image dégradée, alors que ceux qui disposent d'un débit important, peuvent recevoir les images intégrales [Paté 98].

RAMP supporte des niveaux de qualité de service variables selon les destinataires. Pour cela, le protocole supporte deux modes de transmission non fiables. Le premier est un service de type *best-effort*, du même type que celui proposé par UDP. Le second est un service mixte, avec lequel l'expéditeur peut assurer la fiabilité des transmissions vers les seuls destinataires qui le souhaitent.

Par la suite, ce protocole a été optimisé pour une utilisation au-dessus des réseaux optiques à haut débit, de type commutation de circuit, tels que le TBONE (*Testbed for Optical NEtworking*) [Koifman 96]. Avec ce type de réseau, les erreurs de transmission sont rarement à l'origine des pertes de message. La seule véritable cause de ces pertes est la surcharge des tampons de réception des destinataires. Cette propriété a fortement influencé la conception du protocole, qui cherche à minimiser les traitements au niveau des destinataires.

Comme MTP, RAMP utilise une stratégie de retransmission basée sur des acquittements négatifs. Toutefois, cette stratégie est très différente de celle de MTP :

– lorsque le nombre de destinataires n'ayant pas reçu un paquet est faible, les retransmissions de message sont réalisées à l'aide de communications point-à-point plutôt que par rediffusion des paquets vers l'ensemble des destinataires ;

– les demandes de retransmission par les destinataires sont envoyées dès que la perte d'un paquet est détectée.

RAMP propose aux expéditeurs le choix entre deux modes de fonctionnement : un mode *Rafale* (*Burst mode*) et un mode *Inaction* (*Idle mode*). Une rafale est une suite de paquets émis consécutivement, en un temps très court. Le mode *Rafale* oblige les destinataires à envoyer un acquittement positif pour chaque rafale reçue. Avec le mode *Inaction*, l'expéditeur ne doit jamais cesser d'émettre des paquets, même lorsqu'aucune donnée utile n'a besoin d'être envoyée. Durant ces périodes d'inactivité, l'expéditeur est donc obligé d'envoyer des paquets vides à intervalles de temps régulier ; lorsqu'un destinataire constate qu'il n'a pas reçu de données pendant un temps assez long, il sait que des données se sont perdues et envoie un acquittement négatif.

Les auteurs indiquent que le mode *Rafale*, est plus approprié lorsque le nombre de destinataires est faible, car il minimise le volume du traffic diffusé, mais au prix d'un traffic de contrôle accru. Ils préconisent donc le mode *Inaction* lorsque le nombre de destinataires est important. Remarquons, toutefois, que le mode *Inaction* s'applique mal aux réseaux qui ne conservent pas les délais entre les paquets qui sont émis consécutivement. Cette situation est courante sur les réseaux Internet, en particulier lorsque les paquets doivent franchir

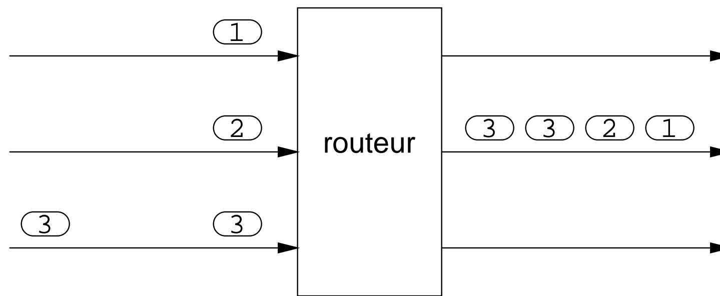


FIG. 3.5: *Compression des délais inter-paquets dans les routeurs*

de nombreux routeurs. Lors du franchissement d'un routeur, certains paquets sont ralentis alors que d'autres passent le routeur sans délai (figure 3.5) [Partridge 93, chapitre 12]. Les paquets émis à intervalles de temps réguliers peuvent donc arriver à destination en rafales. Lorsque la durée qui sépare deux rafales devient trop importante, le destinataire déclenche à tort une demande de retransmission. Le mode *Inaction* est donc particulièrement inadapté lorsque les destinataires sont nombreux et éloignés de la source. Par ailleurs, comme pour XTP, l'extensibilité de ce protocole est limitée en raison de son architecture concentrique.

Cependant, RAMP présente aussi de nombreuses qualités. Il propose des groupes dynamiques et, surtout, un mécanisme pour l'administration de ces groupes et la gestion de leurs identifiants. RAMP est le seul protocole qui ait cherché, à notre connaissance, à apporter une réponse à ce problème. La stratégie de régulation de flux par adaptation du débit est aussi très intéressante lorsque le réseau est très fiable, mais hétérogène, comme c'est le cas avec la majorité des réseaux locaux actuels. Enfin, sa capacité à gérer pour un même flot de données, des niveaux de qualités de service différents, et ce, tant à l'initiative des receveurs qu'à celle de l'expéditeur, est remarquable.

3.3.5 RMTP

Le protocole RMTP (*Reliable Multicast Transport Protocol*) de Lucent Technologies[3] [Paul 97], est un protocole de *multicast* fiable et ordonné (selon la source) prévu pour fonctionner à grande échelle, en particulier sur Internet, au-dessus de IP/Multicast. Ce protocole propose des groupes de communication dynamiques. Ces groupes sont organisés de façon hiérarchique, ce qui permet de tenir compte de la topologie du réseau, et de minimiser le volume du trafic de contrôle généré par le retour des acquittements vers la source de l'émission. En effet, bien que ce protocole utilise une technique d'acquittements positifs, ceux-ci ne "remontent" jamais jusqu'à la source.

La structure hiérarchique de ce protocole reprend le principe de l'arbre de diffusion. Les nœuds de l'arbre sont appelés des *receveurs désignés* (*Designated Receiver*). Chaque receveur désigné conserve les paquets diffusés jusqu'à ce qu'ils aient été reçus par chacun de

3. Il existe un autre protocole multipoints fiable appellé RMTP. Ce protocole est développé par NTT et IBM et propose des groupes de communication statiques.

leurs fils directs (qui sont soit des feuilles, soit eux-mêmes des receveurs désignés). Chaque nœud et feuille de l'arbre envoie régulièrement un message d'état vers son père. Ce message indique au nœud parent quels paquets de la fenêtre courante ont été reçus. Le nœud parent (la racine ou un receveur désigné) retransmet alors les paquets manquants vers ses fils directs. Notons que ce protocole n'implémente pas exactement un arbre de diffusion car les nœuds de ce (pseudo-)arbre de diffusion ne sont pas impliqués dans la transmission initiale du message par la racine ; ils n'en sont que les simples destinataires, au même titre que les feuilles de l'arbre.

Ce protocole implémente un contrôle de flux à l'aide d'une part, d'un mécanisme d'ajustement du débit et d'autre part, d'une technique de fenêtrage. Il intègre un mécanisme de contrôle de congestion à l'aide d'un algorithme de type *slow-start*, comme celui utilisé dans TCP.

Ce protocole présente donc de nombreuses qualités : il utilise une politique de retransmission localisée et sélective, il intègre des mécanismes de contrôle de flux et de contrôle de congestion et son organisation hiérarchique permet de construire des groupes de très grande taille. La politique de retransmission localisée, en particulier, présente plusieurs avantages. Tout d'abord, comme la racine ne reçoit qu'un nombre limité d'acquittements, elle ne souffre pas du phénomène d'inondation. Ensuite, dès lors que les receveurs désignés sont placés suffisamment près de leurs fils, le délai des retransmissions peut être minimisé.

Toutefois, le protocole comporte un défaut. En effet, s'il propose bien un mécanisme pour rattacher dynamiquement un destinataire à son serveur désigné le plus proche, il ne sait pas, en revanche, construire dynamiquement une liste des receveurs désignés ; celle-ci doit donc être construite de façon statique.

Le protocole TMTP [Yavatkar 96] est un autre exemple de protocole hiérarchique, fonctionnant de façon similaire au protocole RMTP.

3.4 Environnements de programmation

Les environnements de programmation répondent aux problèmes de la programmation d'applications réparties, à un niveau purement applicatif, c'est-à-dire sans modification du système d'exploitation. Typiquement, un environnement est constitué d'un certain nombre de programmes de service, qui s'exécutent sur chacun des nœuds de la plate-forme répartie, et de bibliothèques, qui assurent l'interface entre le programme de l'utilisateur et les fonctionnalités offertes par l'environnement.

En général, ces environnements sont rendus disponibles sur la majorité des plate-formes existantes. De fait, les applications qui sont construites à partir de ces environnements sont portables. En contrepartie, cette approche de haut niveau doit se satisfaire des mécanismes proposés dans les systèmes d'exploitation, qui ne sont pas toujours suffisants pour assurer les meilleurs niveaux de performance.

Les environnements de programmation qui proposent des communications multipoints sont étudiés au sein de deux communautés motivées par l'étude problèmes très différents : la communauté de l'algorithmique répartie et la communauté du parallélisme.

La première recherche des solutions aux problèmes généraux posés par la programmation d'applications réparties, sans s'imposer de véritables restrictions quant au contexte d'exécution : l'architecture répartie peut être un réseau local aussi bien qu'un réseau à grande échelle et le réseau comme les machines ne sont pas nécessairement fiables. Le modèle de programmation visé est plutôt le modèle transactionnel ou client/serveur. Toutefois, cette communauté ne s'intéresse aux communications multipoints que pour la construction d'applications tolérantes aux pannes. Au paragraphe 3.4.1, nous présentons une sélection des principaux environnements qui permettent la programmation de telles applications réparties, tolérantes aux pannes.

À l'opposé, la communauté du parallélisme est surtout motivée par l'exploitation des réseaux locaux ou grappes de stations de travail. Le modèle de programmation principalement visé est le modèle SPMD, dans lequel chaque nœud de la plate-forme exécute un même programme, mais avec des données différentes. La plate-forme est donc supposée relativement fiable et les tâches de l'application interagissent en général de façon synchrone, par étape. Les problèmes étudiés sont ceux de la recherche de performances, et en particulier celui de la minimisation du temps d'exécution de l'application. Nous présentons les deux principaux environnements pour la programmation d'applications réparties parallèles au paragraphe 3.4.2.

3.4.1 Environnements pour applications tolérantes aux pannes

Dans ces environnements, la tolérance aux pannes est obtenue par la réplication automatique des données et des programmes. Le niveau de tolérance aux pannes des objets d'une application s'exprime en terme de k-résistance : un objet qui a la propriété d'être k-résistant est un objet dont l'exécution est garantie tant qu'au plus k sites d'exécution tombent en panne. Pour satisfaire cette propriété, un objet doit donc être répliqué sur au moins $k + 1$ sites.

Avec une application répartie traditionnelle (c'est-à-dire non tolérante aux pannes), lorsque deux objets de l'application sont dépendants, et que l'un doit transmettre une information à l'autre, il lui suffit d'envoyer cette information à l'aide d'une communication point-à-point. Dans le cas d'une application tolérante aux pannes, si l'objet destinataire est k-résistant, cette information doit être envoyée à chacune des répliques de l'objet. La nécessité de disposer de mécanismes de communication multipoints dans ce type d'application apparaît donc clairement.

Dans les paragraphes qui suivent, nous décrivons quatre de ces environnements : Isis, Transis, Horus et Totem.

3.4.1.1 Isis

Isis est l'un des premiers environnements à avoir cherché à proposer un support de haut niveau pour la construction d'applications réparties tolérantes aux pannes [Birman 85]. Isis facilite la conception de ces applications en rendant la gestion de la réplication des composants de l'application (objets) transparente pour le programmeur.

Dans l'environnement Isis, les communications multipoints sont réalisées au travers de groupes de communications. Les seuls schémas de communication utilisés sont le *multicast* atomique et la diffusion atomique dans le groupe[4]. La désignation des processus destinataires dans le groupe est explicite. L'environnement Isis propose trois primitives de diffusion/*multicast* atomiques :

- la primitive CBCAST est utilisée pour la transmission des données entre les objets de l'application. Elle implémente un séquencement causal ;
- la primitive GBCAST est utilisée pour la gestion des groupes, et en particulier pour signaler la disparition (panne) de l'un des membres du groupe aux autres membres du groupe. De même que la primitive CBCAST, elle implémente aussi un séquencement causal. Toutefois, cette propriété de séquencement est renforcée de la façon suivante : un GBCAST qui signale une panne est assuré de n'être délivré qu'après la réception de tous les messages envoyés par les processus qui ont disparu du groupe à la suite de la panne ;
- la primitive ABCAST est utilisée pour la transmission des données entre les objets de l'application. Elle implémente un séquencement atomique.

Isis est le premier système à avoir proposé plusieurs formes de séquencement et à avoir introduit la notion de *séquencement virtuel* [Birman 94], au travers du séquencement causal. Cette forme de séquencement est intéressante car elle est moins coûteuse à mettre en œuvre que les formes de séquencement plus strictes, comme le séquencement atomique, et elle offre un niveau de cohérence suffisant pour mettre en œuvre de nombreux algorithmes.

3.4.1.2 Transis

De même que Isis, Transis [Moser 95] est un environnement conçu pour faciliter la conception d'applications tolérantes aux pannes sur des architectures réparties faiblement couplées. De façon similaire, Transis propose des groupes de communication qui permettent la réalisation de schémas de communication de type *multicast*, fiables et ordonnés.

Les communications multipoints de l'environnement Transis [Malki 94] s'appuient sur le protocole Trans [Melliar-Smith 90, Melliar-Smith 91]. Ce protocole implémente la diffusion fiable et ordonnée à partir d'un mécanisme de diffusion non fiable tel que IP/Multicast. Le principe de fonctionnement de ce protocole est de systématiquement combiner l'accusé de réception d'un message reçu à l'étape i avec la diffusion du message de l'étape $i + 1$. À chaque étape, il n'y a donc qu'un seul processus qui envoie un acquittement, ce qui évite le problème d'inondation. Selon le message qui est acquitté à chaque étape, chaque destinataire sait s'il a manqué un message ou non, selon que l'acquittement qui accompagne le message reçu correspond à un message déjà reçu ou non. Lorsqu'un destinataire s'aperçoit qu'il n'a

4. Dans [Birman 86], les auteurs emploient la même terminologie (*broadcast*) pour désigner un envoi vers un sous-ensemble des membres d'un groupe et un envoi vers la totalité des membres du groupe.

pas reçu un message dont il reçoit l'acquittement, il demande une retransmission en envoyant un acquittement négatif.

L'autre trait caractéristique de Transis est sa capacité à supporter les partitions du réseau. Une partition est le résultat d'une panne (transitoire ou permanente) qui isole les composants d'une application répartie au sein de plusieurs sous-groupes indépendants. Dans une telle situation, deux stratégies sont envisageables. La plus simple est celle adoptée par l'environnement Isis : une partition primaire est désignée et les autres partitions sont définitivement invalidées. Cette stratégie est donc difficilement compatible avec un réseau dans lequel les pannes ou pertes de connexion sont fréquentes, comme c'est le cas par exemple avec les réseaux mobile. À l'inverse d'Isis, Transis permet à plusieurs partitions de continuer à s'exécuter de façon indépendante. Il permet de plus le regroupement de ces partitions lorsque la connectivité est rétablie.

Transis propose quatre niveaux de séquencement. Par ordre croissant de garantie, il s'agit des séquencements FIFO selon la source, causal, causal atomique et causal atomique sécurisé. Les deux derniers niveaux diffèrent de la façon suivante : en cas de perte de connexion, le séquencement non sécurisé s'autorise à délivrer les messages à l'application même s'ils n'ont pas été reçus par tous les destinataires. Avec le séquencement sécurisé, un message ne peut pas être délivré à l'un de ses processus destinataires tant qu'il n'a pas été reçu (c'est-à-dire acquitté) au niveau de chacun de ses destinataires.

3.4.1.3 Horus

Horus [van Renesse 96] est le successeur de Isis. Les améliorations apportées concernent l'architecture et la flexibilité du système. En effet, la forme de séquencement la moins coûteuse proposée par Isis, le séquencement causal, reste encore trop coûteuse dans certaines situations. C'est le cas par exemple lorsque le réseau est fiable. Dans ce cas, l'acquittement systématique des messages devient inutile. Par ailleurs, certaines fonctionnalités, comme le cryptage systématique des communications dans un environnement non fiable, ne sont pas disponibles dans Isis.

La stratégie retenue dans le système Horus reprend le principe des STREAMS proposés dans les UNIX de la famille Système V [Padovano 93]. Cette stratégie consiste à proposer des modules indépendants, implémentant chacun une fonctionnalité de communication spécifique : acquittement, cryptage, fragmentation, contrôle de flux, etc. Ces modules peuvent être empilés à volonté, comme les modules du mécanisme des STREAMS. Selon le contexte dans lequel l'utilisateur exécute son application, il choisit l'ensemble des modules qui doivent être combinés pour la réalisation des communications.

Les niveaux de qualité de service supportés par Horus incluent, notamment, un mode *best-effort*, et des modes fiables avec différents niveaux de séquencement (FIFO selon la source, causal, synchrone). Horus reprend, de plus, les améliorations apportées par le système Transis en matière de partitionnement du réseau. L'environnement permet aux partitions d'une application de s'exécuter de façon indépendante en cas de perte de connectivité. Il permet aussi le regroupement de ces partitions lorsque la connectivité est rétablie.

3.4.1.4 Totem

De même que les environnements précédents, l'environnement Totem [Dolev 96] est conçu pour offrir des groupes de *multicast* qui permettent la construction d'applications tolérantes aux pannes. Les autres motivations importantes qui ont guidé l'élaboration de ce système sont la recherche de performances et le support temps réel. Totem vise les applications réparties conçues pour s'exécuter dans l'environnement d'un réseau local. Totem propose un mécanisme de *multicast* totalement ordonné, qui exploite l'éventuelle capacité de diffusion du réseau sous-jacent.

De même que Transis et Horus, Totem supporte les partitions du réseau. L'environnement Totem se distingue des précédents par le protocole qu'il utilise pour obtenir un ordre total sur les messages diffusés. Totem utilise pour cela un protocole à base de jeton : seule la machine qui détient le jeton peut lancer une diffusion. Le jeton circule de machine en machine à l'aide de communications point-à-point fiables.

3.4.2 Environnements pour applications parallèles

L'objectif de ces environnements est de fournir l'ensemble des outils qui permettent aux utilisateurs de concevoir des applications parallèles qui puissent s'exécuter sur un réseau de stations de travail, de la même façon que si elles devaient s'exécuter sur une machine parallèle à mémoire répartie.

Dans les paragraphes qui suivent, nous présentons d'abord PVM. PVM est remarquable en ce sens qu'il a réussi à imposer un standard de fait. Bien que loin d'être parfait, cet environnement a simplement su répondre très tôt et de façon suffisamment efficace aux besoins d'une communauté importante d'utilisateurs.

Nous présentons ensuite MPI. MPI résulte d'un effort de normalisation impliquant d'un côté les principaux constructeurs de machines parallèles, et de l'autre un ensemble d'institutions et d'utilisateurs. PVM et MPI représentent aujourd'hui les deux principaux standards pour la programmation d'applications parallèles par passage de messages, sur les réseaux et grappes de stations de travail.

3.4.2.1 PVM

PVM (Parallel Virtual Machine) [Geist 94] est un système générique pour la programmation d'applications parallèles communiquant par échanges de messages. L'environnement PVM est formé de bibliothèques et de processus de service. Les bibliothèques (l'une pour la programmation en langage “C” et l'autre pour la programmation en langage fortran) réalisent l'interface entre les différents programmes d'une application répartie et le système PVM. Les processus de service s'exécutent sur chacun des nœuds de la plate-forme. Ils prennent en charge la gestion des processus de l'application et tout ou partie de leurs communications. L'objectif du système PVM est de permettre aux applications d'exploiter de façon transparente les nœuds d'une ou plusieurs architectures réparties hétérogènes.

La grande simplicité d'utilisation de ce système, ainsi que sa robustesse et sa gestion transparente de l'hétérogénéité l'ont rendu très populaire dans la communauté du parallélisme. À ce titre, PVM est certainement l'acteur principal de la transition opérée par cette communauté, du monde des machines parallèles vers celui des réseaux de stations de travail. Initialement conçu pour exploiter les stations de travail fonctionnant sous UNIX, PVM a, par la suite, été adapté sur un grand nombre d'architectures parallèles.

PVM est conçu pour donner l'illusion aux applications qu'elles s'exécutent sur une machine parallèle virtuelle. Le système se limite donc au mode de fonctionnement mono-utilisateur des machines parallèles : lorsque plusieurs utilisateurs veulent exécuter des applications PVM sur un même réseau de stations de travail, celles-ci s'exécutent de façon indépendante, sur des machines parallèles virtuelles indépendantes. PVM assure la gestion des tâches de l'application et peut, en particulier, prendre en charge leur placement de façon automatique sur les nœuds de la machine parallèle virtuelle. En revanche, il n'intègre aucun mécanisme de répartition de charge (le mécanisme de placement utilise une stratégie cyclique, indépendante de l'état de charge des machines).

PVM propose des primitives de communication point-à-point et multipoints fiables, asynchrones, et ordonnées selon la source. Le système permet la réalisation de communications multipoints par adressage explicite ou par l'intermédiaire de groupes de communication. Les groupes de communication sont dynamiques et ouverts (aux seules tâches de l'application) : une tâche qui n'appartient pas à un groupe peut envoyer un message dans le groupe. Des primitives sont proposées pour déterminer la composition des groupes. La gestion des groupes est centralisée.

Les schémas de communication multipoints proposés sont la diffusion dans un groupe, le *multicast* par adressage explicite, la barrière de synchronisation et la réduction. Les opérateurs de réduction supportés sont le minimum, le maximum, la multiplication et la somme. Ils peuvent être appliqués à des entiers, des flottants et des complexes. Les primitives d'émission et de réception sont bloquantes, mais une primitive de test est proposée afin de déterminer si un message est en attente de lecture.

D'un point de vue implémentation, la diffusion et le *multicast* dans PVM n'ont pas fait l'objet d'optimisation. La diffusion dans un groupe, en particulier, est très inefficace, puisqu'elle est réalisée en deux temps. Une première communication est établie avec un nœud serveur de groupe afin d'obtenir la liste des membres du groupe. Puis, la fonction de *multicast* à adressage explicite est appelée avec la liste des destinataires reçue de ce serveur. Le *multicast* à adressage explicite est réalisé selon la technique du *multi-unicast*. Notons toutefois que des implémentations optimisées sont généralement proposées sur les machines parallèles. Sur les réseaux de stations de travail, certaines optimisations exploitant des protocoles de *multicast* fiables et ordonnés au dessus de IP ont été proposées, comme celle de [Hall 94].

3.4.2.2 MPI

MPI (*Message Passing Interface*) [MPI Forum 96] est le résultat d'un effort de standardisation en matière d'outils pour la programmation d'applications parallèles réparties dont

les tâches communiquent par passage de messages. La définition de ce standard a impliqué une soixantaine de personnes, regroupées au sein du *Forum MPI*, et provenant d'une quarantaines d'organisations (universités, laboratoires, industriels), principalement aux États-Unis et en Europe.

L'objectif principal du standard MPI est de faciliter la conception et la portabilité des applications parallèles : des implémentations du standard sont proposées sur la majorité des machines parallèles et stations de travail. Une application écrite sur une plate-forme, pour un outil qui respecte la norme est donc *théoriquement* portable sans modification sur toutes les plate-formes où des outils conformes sont proposés. Théoriquement, car la norme définit tellement de fonctionnalités qu'en pratique, les outils proposés sur chaque plate-forme ont souvent du mal à les implémenter toutes.

En effet, la démarche adoptée par le Forum MPI a été d'intégrer au standard les caractéristiques jugées les plus intéressantes des principaux environnements de programmation existant alors. Plus précisément, les objectifs que se sont fixés les membres du Forum MPI sont les suivants :

– définir une interface de programmation (API) ;
– proposer des mécanismes de communication efficaces et performants, qui évitent les recopies en mémoire des messages, et permettent le recouvrement des phases de calcul et de communication ;
– permettre un fonctionnement en environnement hétérogène ;
– fournir une API pour les langages "C" et fortran77 ;
– proposer un mécanisme de communication fiable ;
– définir une API qui soit proche des interfaces existantes ;
– définir une API qui puisse être implémentée sur la majorité des plate-formes existantes sans modification majeure ;
– rendre l'API indépendante du langage de programmation ;
– permettre à l'API de supporter les fils d'exécution multiples (*threads*).

Le standard définit les éléments suivants :

– les communications point-à-point ;
– les opérations collectives : toutes les opérations que nous avons présentées au paragraphe 3.2.3, plus une opération de réduction-scatter, la barrière de synchronisation et l'opération de réduction préfixe (*scan*) ;
– la gestion de groupes de processus : les groupes de communication sont utilisés pour définir les ensembles ordonnés de processus sur lesquels s'appliquent les opérations collectives (chaque processus membre d'un groupe est numéroté) ;

- la gestion de contextes de communication : les contextes de communication sont une abstraction qui permet d'isoler les les communications réalisées par les multiples composants de bibliothèques qui peuvent être utilisés au sein d'une application ;
- la définition de topologies virtuelles de processus : les topologies virtuelles permettent d'identifier les processus selon des schémas plus élaborés que la numérotation linéaire utilisée dans les groupes. Elles permettent de plus d'aider au placement des tâches sur la topologie physique de l'architecture ;
- la définition de primitives pour les langages "C" et fortran77 ;
- la gestion et la récupération d'informations d'environnement, comme le numéro de tâche sur un nœud ou l'existence d'une horloge globale commune à tous les processeurs ;
- une interface pour l'optimisation des programmes.

Sur les réseaux de stations de travail, plusieurs implémentations de la norme MPI ont été proposées, comme LAM [Burns 94] ou MPICH [Gropp 96].

3.5 Systèmes d'exploitation répartis

En dépit de leur nombre impressionnant, rares sont les systèmes d'exploitation répartis pour réseaux de stations de travail qui proposent des mécanismes de communication par passage de message multipoints. Nous présentons les deux plus classiques, Amoeba et Chorus.

3.5.1 Amoeba

Amoeba [Mullender 90, Kaashoek 92, Tanenbaum 95] est un système réparti fortement couplé (voir paragraphe 2.2, page 17), construit au dessus d'un micro-noyau. Ce système est optimisé pour les architectures réparties formées à partir de *bancs de processeurs* (*processor pool*), de serveurs spécialisés et de terminaux graphiques. Les bancs de processeurs sont des collections de nœuds comportant chacun un processeur, une mémoire volatile propre et une interface réseau.

À la différence des nœuds d'une grappe de stations de travail, les nœuds d'un banc de processeurs peuvent ne pas fonctionner de façon autonome. Ils sont normalement placés dans un même boîtier ou une même armoire, et se partagent certains accessoires, comme l'alimentation. Les serveurs spécialisés sont, par exemple, des serveurs de fichiers, des serveurs de nom ou encore des serveurs de périphériques. Les terminaux graphiques ne sont pas supposés capables d'exécuter des applications.

Amoeba supporte l'hétérogénéité de façon transparente et intègre un mécanisme de répartition dynamique de charge par placement. Amoeba n'est que partiellement compatible avec UNIX (compatible avec la norme POSIX).

Amoeba propose deux mécanismes de communication : un mécanisme point-à-point d'appel de procédure à distance, et un mécanisme multipoints reposant sur des groupes

de communication. Les groupes de communication sont fermés et dynamiques. Des primitives sont proposées pour connaître la composition d'un groupe. Le seul schéma de communication multipoints supporté est la diffusion fiable, atomique, et totalement ordonnée dans un groupe de communication.

Pour réaliser ses communications, Amoeba implémente son propre protocole de niveau réseau (c'est-à-dire au même niveau que IP) : FLIP (*Fast Local Internet Protocol*). Ce protocole exploite les capacités de diffusion ou de multicast des réseaux sous-jacents, lorsqu'elles sont disponibles. Par ailleurs, ce protocole intègre un mécanisme d'authentification des destinataires à base de clef publique et de clef privée. La clef publique est calculée à partir d'une fonction de hachage non réversible de type DES (*Data Encryption Standard*). La clef publique représente l'adresse d'un processus (c'est-à-dire l'adresse qu'utilisent les autres processus pour lui envoyer un message). Pour que les messages envoyés à un processus lui soient délivrés par le système, il faut que ce processus ait, au préalable, prouvé son identité au système en lui donnant sa clef privée.

3.5.2 Chorus

Chorus [Rozier 88, Rozier 90, Tanenbaum 95] est un système d'exploitation réparti lui aussi basé sur un micro-noyau. Chorus offre des fonctionnalités avancées, comme la compatibilité binaire avec UNIX (compatibilité avec les UNIX de type Système V), un support d'exécution pour les applications temps réel et un mécanisme de mémoire virtuelle partagée.

Chorus propose deux modèles de communication par passage de message : des communications par envoi de message asynchrone et des communications par appel de procédure à distance (RPC). Les communications de type RPC sont fiables, à l'inverse des communications par envoi de message asynchrone. Ces deux formes de communication s'appuient sur un mécanisme de type "boîte à lettres", appelé *port*. Les ports sont créés par les processus qui désirent recevoir des messages. Chaque processus peut créer plusieurs ports selon ses besoins. Ces ports peuvent migrer d'un processus à un autre, mais à un instant donné, chaque port n'est associé qu'à un unique processus. Le système attribue à chaque port un identifiant global unique (dans l'espace et dans le temps). La localisation des ports (et de leur processus propriétaire) n'a donc pas besoin d'être connue par les émetteurs. Les messages reçus sur un port sont conservés jusqu'à ce que leur propriétaire les consomme. Les ports ont une capacité de stockage limitée. Lorsque la limite de cette capacité est atteinte, les processus émetteurs sont bloqués.

Chorus propose des groupes de communication au travers d'une abstraction, le *groupe de port* (*port group*). Comme son nom l'indique, un groupe de port permet de regrouper plusieurs ports. Un port peut faire partie de plusieurs groupes. Le processus qui crée un groupe récupère un descripteur lui donnant la capacité de gestion sur le groupe. Ce descripteur peut ensuite être utilisé pour ajouter ou supprimer des ports dans le groupe. Ce descripteur peut être partagé avec d'autres processus ; seuls les processus qui ont accès à ce descripteur ont le droit d'ajouter ou supprimer des ports dans le groupe. Cette capacité de gestion est indépendante de la capacité d'envoyer des message dans le groupe. De la même façon que les

ports, les groupes de ports possèdent un identifiant global unique. Tous les processus peuvent utiliser cet identifiant pour envoyer des messages au groupe.

L'envoi d'un message dans un groupe ne peut être obtenu qu'avec le mode de transmission asynchrone non fiable. Chorus décline quatre formes de communication multipoints à partir des groupes de ports :

- l'envoi d'un message à destination de tous les ports du groupe (diffusion) ;
- l'envoi d'un message à destination de l'un des ports du groupe. Dans ce cas, le système propose à l'application trois stratégies de choix :
  - le choix arbitraire d'un port du groupe ;
  - le choix de l'un des ports du groupe qui ne se trouve pas sur un nœud donné ;
  - le choix de l'un des ports du groupe qui se trouve sur un nœud donné.

3.6 Conclusion

Chacune de ces quatre communautés aborde les communications multipoints de façon différente, en fonction de ses préoccupations immédiates. Certains problèmes, qui se posent dans chacun des domaines, conduisent à des solutions qui font l'unanimité, comme l'utilisation des groupes de communication pour répondre au problème de l'adressage multipoints, par exemple.

Cependant, la majorité des problèmes étudiés n'est spécifique qu'à l'une, voire quelques unes de ces quatre communautés et, par conséquent, généralement ignorée dans les solutions proposées au sein des autres.

Les environnements pour la programmation d'applications tolérantes aux pannes, par exemple, s'intéressent aux problèmes de séquencement, mais ils ne cherchent pas à élaborer des schémas de communication plus complexes que celui de la diffusion. À l'opposé, les environnements pour le parallélisme se satisfont d'un séquencement FIFO selon la source, mais proposent des schémas de communication élaborés.

Par ailleurs, la plupart des environnements de programmation proposent des mécanismes qui permettent de faciliter la mise au point et le déboguage des applications réparties. Ces mécanismes ne sont que rarement considérés au niveau de l'étude des systèmes d'exploitation et encore moins au niveau des protocoles. En contrepartie, les environnements de programmation imposent certaines contraintes d'utilisation dont n'ont pas à souffrir (en principe) les autres solutions. Ils imposent, en particulier, un mode et un langage de programmation, à l'inverse des systèmes d'exploitation et des protocoles.

De leur côté, les systèmes d'exploitation sont prévus pour gérer efficacement les flots de communication émanant d'applications concurrentes, alors que bon nombre d'environnements de programmation ne sont conçus que pour un fonctionnement mono-application. Les systèmes d'exploitation autorisent, de plus, une implémentation optimale de certaines

fonctionnalités ou services, car ils ont la possibilité de les exécuter dans le contexte privilégié du noyau. En revanche, toute application qui exploite les mécanismes de communication spécifiques d'un système d'exploitation se trouve, de ce fait, confrontée à un problème de portabilité vers les autres systèmes d'exploitation.

Contrairement aux solutions précédentes, les protocoles ne proposent que des solutions théoriques. Leur déploiement se heurte donc à un problème de disponibilité. Afin d'être validés, les protocoles que nous avons présenté ont bien sûr fait l'objet d'implémentations. Mais ces dernières ne sont souvent réalisées qu'en contexte utilisateur. Or, pour être la plus efficace possible, la mise en œuvre d'un protocole doit s'opérer au niveau du système d'exploitation.

Bien sûr, une implémentation en contexte utilisateur d'un protocole multipoints offre toujours de meilleurs résultats que lorsque les mêmes schémas de communication sont réalisés à partir de simples communications point-à-point. On peut donc s'interroger sur la réelle nécessité d'une mise en œuvre au niveau du système d'exploitation, qui est beaucoup plus délicate. Dans le cas de réseaux aux performances modestes, comme Ethernet à 10 Mbits/s, elle n'est pas clairement établie. En revanche, pour des réseaux à hautes performances comme FastEthernet, ATM ou Myrinet, ce niveau de mise en œuvre se justifie pour au moins deux raisons :

- le coût de traitement des messages au niveau des couches de protocole devient prépondérant par rapport à la latence physique du réseau. Avec un réseau Myrinet, par exemple, la latence au niveau physique n'est que de quelques micro-secondes, alors qu'elle atteint globalement plusieurs centaines de micro-secondes lorsque le réseau est utilisé au travers de la suite de protocoles TCP/IP. Une mise en œuvre en contexte utilisateur se traduit donc par une augmentation conséquente de cette latence, car elle introduit des opérations coûteuses sur le chemin critique de traitement du message (recopies du message, changements de contexte) ;
- plus le réseau est rapide, plus le débit des émissions et réceptions de messages s'accroît et plus le système doit consacrer du temps à la gestion des messages. Et là encore, cette durée peut prendre des proportions considérables. Par exemple, nous avons mesuré que sur un PC Linux équipé d'un processeur PentiumPro cadencé à 200 MHz, une tâche qui communique en permanence au travers d'un réseau Myrinet, occupe à elle seule 80% du temps du processeur, alors que cette occupation est à peine mesurable lorsqu'elle communique au-dessus d'un réseau Ethernet. En minimisant le temps de prise en charge des messages, on augmente donc automatiquement la disponibilité du système.

De plus, quel que soit le niveau de sa mise en œuvre, l'optimisation d'un protocole jugé théoriquement très efficace est d'autant plus longue que le protocole est complexe. L'expérience acquise au sein de la communauté Internet avec le protocole TCP/IP en est le parfait exemple : les premières implémentations de TCP/IP, dans le système 4.2BSD, parvenaient difficilement à atteindre 10% de la bande passante maximale d'un réseau Ethernet à 10 Mbits ([Stevens 90, chapitre 17]) ; aujourd'hui bien que les performances des implémentations de

ce protocole aient été très nettement améliorées, TCP/IP continue de faire l'objet de nombreux travaux de recherche.

Cette difficulté de mise en œuvre explique certainement pourquoi un protocole tel que XTP n'a jamais réussi à s'imposer aux côtés de TCP et d'UDP. En effet, si plusieurs implémentations expérimentales de ce protocole on été proposées en contexte utilisateur [Dabbous 93, Strayer 94], il n'existe aujourd'hui, à notre connaissance, qu'une seule implémentation de ce protocole qui soit prévue pour fonctionner dans le contexte d'exécution d'un noyau UNIX. Cette implémentation est un produit commercial, construit au-dessus du mécanisme des STREAMS, et s'adresse donc plutôt aux UNIX de la famille Système V [Mentat 94]. Cette exploitation commerciale en restreint donc malheureusement la diffusion.

Par ailleurs, implémenter un protocole n'est pas suffisant. Il faut aussi concevoir les interfaces de programmation (API) qui permettent d'exploiter ces protocoles au sein d'une application. Les API de programmation existantes dans les systèmes UNIX, et en particulier l'API des *sockets*, semblent bien mal adaptées pour supporter les nouvelles et nombreuses fonctionnalités qui apparaissent ou vont apparaître avec les protocoles multipoints. La solution consistant à fournir une bibliothèque ou un environnement qui permette l'accès aux fonctionnalités du protocole est bien sûr toujours envisageable. C'est d'ailleurs la solution pratique qui a été retenue dans la majorité des protocoles que nous avons présentés. Mais comme nous l'avons vu avec les environnements de programmation, cette solution manque de flexibilité car elle impose un mode et un langage de programmation particuliers.

Deuxième partie

Contributions

Chapitre 4

Construction d'indicateurs de charge multi-critères en environnement hétérogène

4.1 Introduction

Dans ce chapitre, nous présentons une méthodologie pour la modélisation empirique automatisée des éléments d'un réseau de stations de travail, ainsi que l'outil que nous avons développé pour la mettre en œuvre, *LoadBuilder*. L'objectif de ce travail est de permettre la construction d'un ensemble d'indicateurs de charge utiles et significatifs pour la répartition dynamique de charge, dans le contexte fortement hétérogène des réseaux de stations de travail.

4.1.1 Motivations

Comme nous l'avons vu au chapitre 2, les stratégies de répartition dynamique de charge doivent chercher à exploiter au mieux les informations disponibles, afin de trouver, pour chaque tâche, un site qui minimise son temps d'exécution. La façon d'aborder ce problème dépend beaucoup du contexte dans lequel on cherche à le résoudre.

Ainsi, les premières solutions proposées visaient plutôt un contexte dans lequel les machines, encore coûteuses, étaient relativement peu puissantes, et n'étaient disponibles qu'en nombre relativement restreint. Le processeur constituait donc souvent la ressource critique de ces machines. Elles étaient partagées entre de nombreux utilisateurs, dont le comportement était (et reste toujours) imprévisible. Par conséquent, elles pouvaient subir des variations de charge importantes et inattendues. Par ailleurs, les techniques de programmation étaient séquentielles et n'exploitaient pas le réseau de communication de façon intensive. Les programmes étaient donc écrit pour s'exécuter directement au-dessus du système d'exploitation, c'est-à-dire sans s'appuyer sur un environnement d'exécution intermédiaire. Le mécanisme proposé devait donc pouvoir s'intercaler de façon transparente entre ces programmes et le système d'exploitation [Ferrari 88], voire directement au cœur du système d'exploitation [Barak 85]. En conséquence, ce mécanisme ne pouvait espérer aucune information précise, de la part de l'application, concernant les ressources qui allaient être utilisées. Le mieux que l'on pouvait faire était donc de tenter de les prédire d'après l'historique des exécutions précédentes [Devarakonda 89].

Avec l'apparition des super-calculateurs parallèles, on s'est intéressé à la répartition de charge dans un contexte totalement différent [André 88, Bouvry 93]. En effet, ces machines sont prévues pour fournir, à un instant donné, le maximum de performances à un *unique utilisateur*. De plus, les nœuds de calcul de ces machines sont généralement tous identiques. Mais surtout, le modèle de programmation d'une application parallèle est radicalement différent du précédent. L'objectif du parallélisme est en effet de décomposer le traitement d'un problème de grande taille en plusieurs parties, pour traiter chacune d'elles de façon concurrente. En général, chaque nœud exécute un même programme sur une partie des données (paradigme SPMD [Rumeur 94]). Qui plus est, le processeur n'est plus la seule ressource critique pour ces programmes. Les instances de ces derniers sur chacun des nœuds peuvent en effet communiquer entre elles de façon intensive et/ou avoir massivement recours aux entrées/sorties.

Le problème de répartition dynamique qui se pose avec ce type d'application sur ce type de machines n'est donc plus seulement (voire plus du tout) celui de la répartition des

processus, mais aussi celui de la répartition des données entre les tâches d'une même application. En d'autres termes, dans ce contexte, la répartition dynamique de charge n'est plus inter-application, mais intra-application. Ces différences ont des conséquences importantes au niveau de la gestion des informations. Tout d'abord, la répartition de charge doit impérativement tenir compte des communications et des entrées/sorties réalisées par chacune de ces tâches. Ensuite, puisqu'il s'exécute au cœur de l'application, le mécanisme de répartition de charge peut avoir accès à certaines informations concernant les ressources qui vont être utilisées par chacune des tâches de l'application. Enfin, le mécanisme de répartition de charge n'a plus besoin de réagir à des variations de charge inattendues, c'est-à-dire provenant de tâches qui ne sont pas placées sous son contrôle, puisqu'il contrôle chacune des tâches de l'application.

Enfin, ces dernières années, grâce à l'évolution rapide des performances des composants de grande série, les réseaux de stations de travail se sont révélés être suffisamment compétitifs pour servir de plate-forme d'exécution aux applications très gourmandes en ressources, et notamment aux applications parallèles. Par ailleurs, le faible coût des matériels permet d'une part, de renouveler ou de faire évoluer ces derniers plus rapidement, et d'autre part, d'équiper de façon systématique chaque poste de travail d'une machine pratiquement autonome, offrant à son utilisateur une puissance de calcul confortable. Cette évolution a donc deux conséquences remarquables :

1. les réseaux de stations de travail se sont agrandis et sont devenus fortement hétérogènes ;

2. chaque utilisateur dispose de sa propre machine, sur laquelle il dispose d'une puissance largement suffisante, en général, pour traiter l'ensemble de ses applications courantes. Par conséquent, la répartition de la charge issue d'applications séquentielles ne semble plus être une priorité.

Ces évolutions remettent donc en question tout ou partie des stratégies de répartition de charge utilisées dans les contextes précédents. Tout d'abord, parce que les stratégies initialement étudiées dans le cadre des stations de travail n'ont pas été conçues pour les applications parallèles et réparties. Avec ces dernières, les processus sont interdépendants, et généralement pris en charge par un environnement d'exécution qui leur donne l'illusion de s'exécuter sur une machine parallèle virtuelle [Geist 94]. Et inversement, parce que les stratégies conçues pour les machines parallèles ne s'appliquent plus lorsque les applications parallèles sont exécutées sur les réseaux de stations de travail, en raison de leur hétérogénéité et de leur fonctionnement multi-utilisateurs et multi-tâches (les stations de travail fonctionnent généralement sous UNIX).

L'hétérogénéité n'est pas un élément nouveau dans les réseaux de stations de travail. En revanche, il a pris de l'importance et ne peut plus être ignoré, en particulier en ce qui concerne les applications parallèles. Ses conséquences apparaissent ainsi à tous les niveaux : au niveau des performances des machines, avec des architectures dont les performances sont très variables selon les types de traitements demandés (calculs flottants ou entiers, utilisation

intensive des entrée/sorties, de la mémoire, du réseau, etc) ; au niveau des communications, avec des réseaux aux caractéristiques très variables ; au niveau des systèmes d'exploitation, avec des coûts de gestion des ressources et des processus parfois très différents ; et au niveau des configurations, avec des variations importantes à la fois dans l'espace (c'est-à-dire d'une machine à l'autre) et dans le temps (l'ajout ou le remplacement de composants mémoire, de processeurs, de disques durs, ou de cartes d'entrée/sorties, sont des opérations assez courantes, car très faciles et relativement peu onéreuses).

4.1.2 Architecture visée

Le modèle abstrait d'architecture que nous considérons reflète donc cette hétérogénéité : les nœuds de calcul (sites) disposent de leur propre mémoire (modèle MIMD) et d'un système d'exploitation autonome, multi-tâches et multi-utilisateurs (généralement UNIX). Ce système permet aux utilisateurs d'accéder indépendamment à chacun des sites et permet l'exécution distante de tâches depuis un site vers les autres sites, au travers du réseau local. Toutefois, la compatibilité binaire des exécutables entre chacun des sites n'est pas garantie. Chaque site dispose localement de sa propre mémoire de masse, mais celle-ci peut-être utilisée par les autres sites au travers du réseau (par exemple au travers d'un système de fichiers réparti tel que NFS). Les caractéristiques du réseau local (topologie, performances, technologie) sont variables à la fois dans l'espace et le temps. La variabilité dans l'espace implique que les performances des communications entre deux sites sont variables selon les couples de sites. La variabilité dans le temps est néanmoins relativement faible : les caractéristiques du réseau peuvent être supposées constantes au moins pour la durée de vie d'une application et le sont avec une forte probabilité d'une exécution à la suivante.

4.1.3 Applications ciblées

Le modèle d'application qui nous intéresse est celui des *applications réparties communicantes*. Ces applications sont formées de multiples tâches, qui peuvent s'exécuter sur des sites différents et qui peuvent avoir besoin de s'échanger des informations. Une application peut créer de nouvelles tâches au cours de son déroulement. L'exécution de l'application débute par le lancement de son premier processus, sur un site quelconque, et se termine par la fin de son dernier processus, sur un site éventuellement différent. Sur chaque site, les processus d'une application répartie sont en concurrence pour l'accès aux ressources du site (i) entre-elles, (ii) avec les processus d'autres applications (qui ne sont pas nécessairement des applications réparties) et (iii) avec le système d'exploitation[1].

1. Certaines activités du système d'exploitation, comme l'ordonnancement des processus, la gestion des tampons de mémoire des disques, le traitement des interruptions, ou le traitement au niveau des couches de protocole des messages en provenance d'autres sites, entraînent une consommation de ressources qui ne peut pas être attribuée à une tâche en particulier.

4.1.4 Objectif

L'objectif de ce travail est donc de proposer une méthode et des outils permettant la construction automatique d'indicateurs de charge multi-dimensions et multi-critères en environnement hétérogène. En effet, comme nous l'avons vu au chapitre 2, ce type d'indicateurs est devenu indispensable pour permettre la répartition dynamique de la charge des applications parallèles et réparties dans le contexte fortement hétérogène des réseaux et grappes de stations de travail. Les multiples critères sont indispensables car la ressource de calcul (le processeur) n'est plus la seule ressource critique pour ces applications. Les multiples dimensions permettent de prendre en compte efficacement les informations que sont susceptibles de fournir a priori ces applications au mécanisme de répartition charge.

4.2 Méthodologie

Avant de décrire la méthodologie de construction de ces indicateurs, nous allons dans un premier temps préciser les hypothèses qui ont été faites concernant les propriétés du mécanisme de répartition de charge visé. Dans un deuxième temps, nous décrivons de quelle façon nous envisageons d'exploiter ces indicateurs en présentant une stratégie de décision basée sur l'évaluation d'une fonction de coût.

4.2.1 Hypothèses

Dans ce qui suit, nous posons les hypothèses suivantes :

1. le mécanisme de répartition de charge considéré est un mécanisme de placement. Il n'est donc invoqué que lors de la création d'une tâche ;
2. le mécanisme de répartition de charge ne prend pas en charge l'ensemble des applications. La charge induite par les applications qui ne sont pas gérées est appelée *charge d'arrière plan* (ou *charge exogène*) ;
3. le mécanisme de répartition de charge offre la possibilité aux applications prises en charge de lui communiquer des informations a priori concernant les ressources utilisées par chacune de leurs tâches ;
4. les applications ne sont pas supposées être capables de donner une estimation du temps d'exécution de leurs tâches. Elles sont en revanche supposées capables d'identifier les tâches dont la durée de vie est suffisamment longue pour que leur placement sur un autre site soit (potentiellement) intéressant.

4.2.2 Fonction de coût

Le temps d'exécution d'une application répartie représente la durée qui s'écoule entre le début de sa première tâche et la fin de sa dernière tâche. Par exemple, considérons une application formée de quatre tâches, exécutées chacune par quatre processus `p0` à `p3`, dont

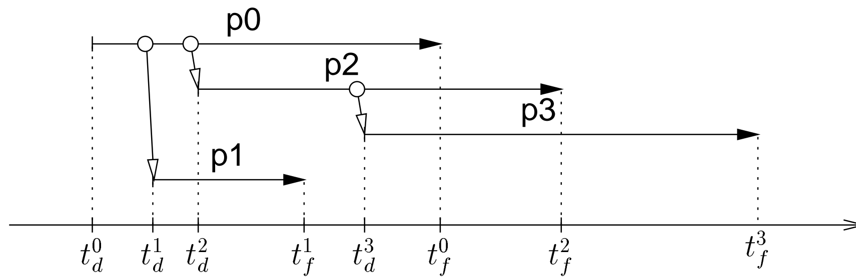


FIG. 4.1: *Exemple de chronologie d'exécution d'une application répartie*

un exemple de chronologie d'exécution est présenté sur la figure 4.1. Dans ce cas, le temps d'exécution de l'application est $T = t_f^3 - t_d^0$. Cependant, l'exemple de chronologie présenté n'est qu'une possibilité parmi d'autres. En effet, en l'absence d'informations a priori concernant la durée d'exécution de chacun de ces processus (ou des synchronisations qui existent entre eux), les dates d'occurrence de leur terminaison ne peuvent pas être ordonnées. Donc la seule chose que l'on puisse affirmer avec certitude est que la date de terminaison de l'application est $T = \max(t_f^1 \cdots t_f^4) - t_d^0$. Atteindre l'objectif fixé avec une stratégie de placement nous conduit donc au problème suivant : à chaque lancement d'une nouvelle tâche, il faut trouver un site d'exécution pour cette tâche qui soit tel que le maximum des dates de terminaison des tâches de l'application soit minimal.

Toutefois, remarquons que lorsqu'une nouvelle tâche se présente, l'ensemble des tâches exécutées par l'application n'est pas supposé être connu. On ne peut donc considérer que les seules tâches actives au moment de la prise de décision. Dans l'exemple présenté sur la figure 4.1, cela nous conduit donc à résoudre les problèmes suivants :

- à $t = t_d^0$: placer p0 de telle façon que t_f^0 soit minimal ;
- à $t = t_d^1$: placer p1 de telle façon que $\max(t_f^0, t_f^1)$ soit minimal ;
- à $t = t_d^2$: placer p2 de telle façon que $\max(t_f^0, t_f^1, t_f^2)$ soit minimal ;
- à $t = t_d^3$: placer p3 de telle façon que $\max(t_f^0, t_f^2, t_f^3)$ soit minimal.

Lorsque les tâches sont interdépendantes, c'est-à-dire lorsqu'elles communiquent entre elles, la résolution de ce type de problème est difficile, car la solution optimale est, en général, le résultat d'un compromis [Folliot 92, Bernon 95, Chatonnay 98]. D'un côté, lorsque deux tâches sont placées sur un même site, elles communiquent entre elles plus rapidement, mais elles sont en concurrence pour l'accès aux ressources de ce site. De l'autre, lorsqu'elles sont placées sur des sites différents, elles n'entrent plus en concurrence pour l'accès aux ressources locales de leur site d'exécution, mais leurs communications sont réalisées moins rapidement. En pratique, la recherche d'un bon compromis peut être obtenue en définissant une fonction de coût qui évalue l'intérêt du placement de la nouvelle tâche sur chaque

site [Bernon 95, Chatonnay 98]. Une telle fonction de coût peut combiner les trois éléments (termes) suivants :

– un terme décrivant la capacité du site à répondre aux besoins de la tâche en dehors de ses phases de communication avec les autres tâches de l'application. Ce terme varie en fonction de la charge du site et des besoins en ressource de la tâche à placer ;

– un terme reflétant le coût des communications sur ce site avec les autres tâches de l'application. Ce terme dépend d'une part de la localisation des autres tâches, du volume des communications avec chacune des autres tâches de l'application, et de la charge du réseau et du site ;

– un terme reflétant le coût lié à la concurrence de la tâche à placer avec les autres tâches présentes sur le site.

4.2.3 Description de la méthodologie

La méthodologie que nous proposons a pour objectif de permettre la construction des éléments d'information nécessaires au calcul de la fonction de coût que nous venons de présenter. Cette méthodologie suppose la définition préalable de l'ensemble des *ressources logiques* que doit distinguer cette fonction de coût. La méthodologie suit alors quatre étapes :

1. construction, sur chaque nœud, d'indicateurs reflétant la contention au niveau de chaque ressource logique ;

2. étalonnage de la puissance de chacun des nœuds vis-à-vis de chaque ressource logique ;

3. modélisation des coûts de communication ;

4. étude sur chaque nœud de l'influence du placement d'une tâche sur les performances d'une autre tâche.

Les deux premières étapes permettent de construire le premier terme de la fonction de coût. Plus précisément, leur but est de construire des indicateurs multi-critères et multi-dimensions qui permettent de comparer les performances de sites hétérogènes (figure 4.2). La troisième étape permet de construire le deuxième terme de la fonction de coût et la dernière étape le troisième.

4.2.3.1 Définition des ressources logiques

Les ressources logiques représentent des catégories d'instructions ou de traitements élémentaires : calculs en nombre flottants ou en nombres entiers, accès à des zones de mémoire contiguës ou distantes, différents types de communication, appels système, créations de processus, etc). Lors de la création d'une nouvelle tâche, l'application est supposée indiquer la proportion moyenne de chaque ressource logique qui va être utilisée par le processus (coefficients α_i sur la figure 4.2). Il reste néanmoins possible d'utiliser une fonction de coût par

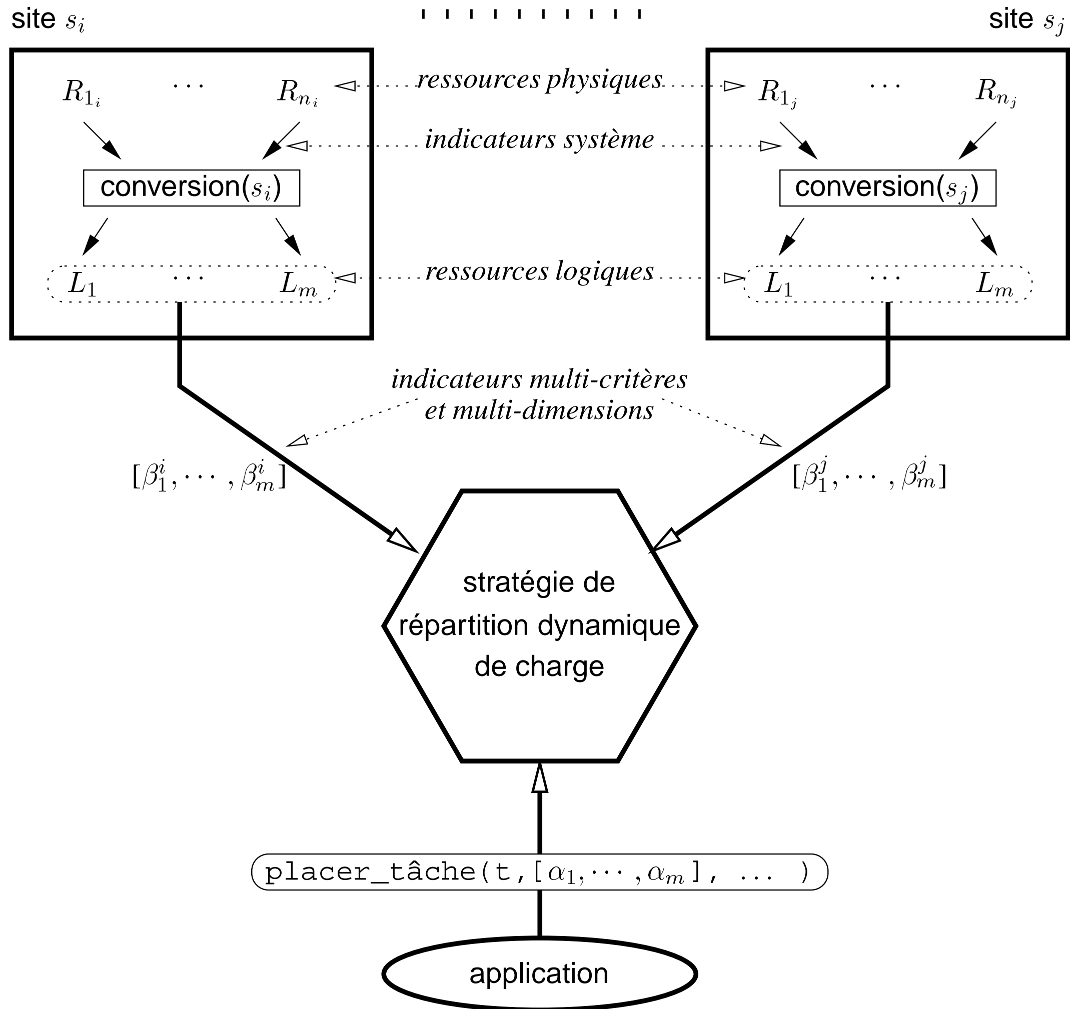


FIG. 4.2: *Définition et utilisation d'indicateurs de charge multi-dimensions et multi-critères en environnement hétérogène.*

Sur chaque site s_i, la charge est exprimée au travers d'indicateurs multi-dimensions ($[\beta_i^1, \cdots, \beta_i^m]$). Chaque dimension représente le *facteur de performance* du site vis-à-vis de l'une des *ressources logiques* considérées par la stratégie de placement. Ce facteur de performance est le produit de la *disponibilité de la ressource* par la *puissance du site vis-à-vis des autres sites*, pour cette ressource. La disponibilité d'une ressource logique est calculée par une fonction de conversion, à partir des informations proposées par le système d'exploitation du site. Lorsqu'une application désire placer une tâche, elle décrit les besoins de cette tâche vis-à-vis de chacune des ressources logiques ($[\alpha_1, \cdots, \alpha_m]$). La capacité d'un site à répondre aux besoins de la nouvelle tâche (premier terme de la fonction de coût présentée au paragraphe 4.2.2) est alors la somme $\sum_{l=1}^{m} \alpha_l \cdot \beta_l^i$.

défaut lorsque ces informations a priori ne sont pas fournies par l'application. On se ramène dans ce cas à la situation d'un placement multi-critères mono-dimension.

Ces ressources sont logiques car leurs relations avec les ressources physiques[2] d'un site ne sont pas bijectives. Par exemple, les ressources logiques de calcul flottant et de calcul entier sont toutes deux liées à la ressource physique du processeur, alors que les ressources logiques de communication sont liées aux ressources physiques du processeur et du réseau. Distinguer de nombreuses ressources logiques permet de mieux prendre en compte l'hétérogénéité des nœuds. Néanmoins, remarquons que multiplier le nombre des ressources logiques rend la description des tâches de l'application plus difficile. En effet, les ressources utilisées par chaque tâche peuvent être soit indiquées de façon explicite par le programmeur, soit apprises à partir d'exécutions précédentes de la tâche [Hémery 94].

4.2.3.2 Indicateurs de charge

Les indicateurs de charge que nous cherchons à construire sont en fait des *indicateurs de disponibilité*. La disponibilité d'une ressource à un instant donné représente le nombre de requêtes pour cette ressource qui peuvent être servies par unité de temps, par rapport au nombre maximal de requêtes qui pourraient être servies en l'absence totale de concurrence pour l'accès à cette ressource. Cette dernière quantité représente la disponibilité maximale de la ressource. La disponibilité à un instant donné peut donc s'exprimer à partir d'un pourcentage de cette disponibilité maximale.

La difficulté d'évaluation de la disponibilité des ressources logiques est très variable. On peut distinguer trois cas de figure :

1. cette disponibilité est exclusivement liée à la disponibilité d'une ressource physique. Par exemple, la disponibilité de la ressource logique de calcul sur des entiers est exclusivement liée à la disponibilité du processeur. Elle est dans ce cas relativement simple à évaluer, à partir de la taille moyenne de la file d'attente des processus prêts à s'exécuter ;

2. cette disponibilité n'est liée qu'à la disponibilité de plusieurs ressources physiques. C'est le cas par exemple des ressources logiques liées aux entrée/sorties sur disque, qui ne dépendent (en principe) que de la disponibilité du processeur et de celles des périphériques concernés. La disponibilité de ces ressources logiques est plus difficile à évaluer, car d'une part, il faut trouver le moyen d'observer la disponibilité de chacune des ressources physiques et, d'autre part, construire un modèle du nombre de requêtes pour la ressource logiques qui sont servies en fonction de de la disponibilité de chacune des ressources physiques. Cette modélisation implique donc d'être capable de faire varier la disponibilité de chacune des ressources physique. La technique de modélisation la plus courante consiste à définir des générateurs de charge synthétique, dont le niveau d'activité pour chacune des ressources physiques peut être paramétré.

---

2. Nous appelons ressources physiques l'ensemble des ressources prises en charge par le système d'exploitation d'un nœud.

3. cette disponibilité n'est pas seulement liée à la disponibilité de ressources physiques, mais dépend d'autres facteurs. C'est le cas par exemple des ressources logiques liées aux communications (qui dépendent de la charge du réseau) ou à la mémoire (qui dépendent notamment des mécanismes de cache, de pagination, et de la façon dont la mémoire est utilisée par la tâche à placer). Dans ce cas, il n'existe malheureusement pas de méthode générale. Dans le cas des ressources liées aux communications, une méthode consiste à modéliser l'influence de la charge du réseau sur les performances des communications. Dans le cas des ressources logiques liées à la mémoire, tout dépend de la nature des ressources logiques considérées. Considérer la disponibilité globale de la mémoire est sans espoir, car cette disponibilité n'est pas uniforme. En revanche, on peut étudier cette disponibilité dans certains cas particuliers d'utilisation. Une stratégie possible consiste à définir un ensemble de ressources logiques correspondant chacune à des manipulations de zones de mémoires de tailles différentes. On peut, par exemple, remarquer qu'un processus qui n'utilise qu'une faible quantité de mémoire ou qui travaille de façon prolongée sur de petites zones de sa mémoire est a priori beaucoup moins sensible aux perturbations liées à la pagination, qu'un processus qui accède de façon aléatoire à de très grandes zones de mémoire, car la mémoire qu'il utilise n'a que très peu de chances d'être paginée. Les facteurs qui ont une influence sur les performances des accès répétés à une petite zone de la mémoire sont ainsi plus facilement identifiables : il s'agit a priori des facteurs qui sont susceptibles d'entraîner de nombreux défauts de cache, comme, par exemple, un nombre important de processus actifs.

4.2.3.3 Étalonnage de la puissance respective de chacun des nœuds

Cette opération est relativement simple et peut être automatisée. Il suffit de construire un programme d'évaluation de performance pour chacune des ressources logiques et d'en comparer les temps d'exécution sur chacun des nœuds de l'architecture. La puissance respective de chacun des nœuds peut ensuite être exprimée par rapport à la puissance de l'une des machines choisie comme référence [Chatonnay 98].

4.2.3.4 Modélisation des coûts de communication

Le coût d'une communication dépend de plusieurs facteurs :

– le trafic d'arrière-plan du réseau de communication ;

– les performances nominales du réseau de communication, c'est-à-dire en l'absence totale de charge (en particulier la latence et la bande passante) ;

– la charge locale des nœuds impliqués dans les communications ;

– le type des communications (longueur des messages, protocole de communication utilisé).

Dans le contexte qui nous intéresse, la seule stratégie raisonnable pour modéliser l'influence de chacun de ces facteurs sur les performances des communications est d'opter pour

une approche de modélisation empirique. En effet, ces facteurs dépendent de nombreux paramètres liés tant à l'architecture et à l'implémentation des systèmes d'exploitation, qu'à l'architecture et aux performances des matériels. Prendre en compte chacun de ces paramètres dans un modèle analytique ou une simulation est donc impossible lorsque le contexte est fortement hétérogène et que les configurations changent régulièrement. La technique de modélisation proposée consiste donc à définir un programme de test que mesure le temps de réalisation d'une communication. Ce programme est exécuté de multiples fois, en faisant varier à chaque fois la valeur des paramètres du modèle. L'analyse statistique des corrélations entre les performances mesurées par ce programme et les variations des paramètres permet de construire le modèle recherché.

4.2.3.5 Influence du placement d'une tâche sur les performances des autres tâches

Là encore, modéliser les conséquences du placement d'une nouvelle tâche est difficilement envisageable autrement que par une démarche empirique, car les paramètres du modèle sont nombreux. En effet, l'influence du placement sur un site s d'une nouvelle tâche t_j sur les performances d'une autre tâche t_i, déjà en cours d'exécution, dépend des trois éléments suivants :

- la charge d'arrière plan du nœud s ;
- la nature des ressource utilisées par le processus exécutant la tâche t_i ;
- la nature des ressources qui vont être utilisées par la tâche t_j.

Chacun de ces trois éléments est une combinaison de plusieurs paramètres : la charge locale dépend de la disponibilité de chacune des ressources logiques (ou physiques) et les ressources utilisées par chacune des tâches s'expriment en fonction des ressources logiques. De la même façon que pour les coûts de communication, la méthode que nous proposons consiste à mesurer les performances obtenues par un programme de test. La performance peut être mesurée en comptant le nombre d'opérations que parvient à réaliser ce programme par unité de temps. Le type des opérations réalisées par ce programme doit pouvoir être paramétré, afin de refléter l'utilisation variable des différentes ressources logiques. La charge d'arrière-plan peut être simulée et contrôlée à l'aide de programme générateur de charge synthétique. Notons bien sûr que, puisqu'un programme qui utilise des ressources génère lui-même de la charge, le programme de mesure peut être utilisé afin d'établir les niveaux de charge synthétique.

4.3 Description de la plate-forme *LoadBuilder*

Comme nous venons de le voir, la méthodologie que nous proposons s'appuie sur une démarche empirique, qui exige la réalisation de très nombreuses expériences. Ces expériences ont pour but d'étudier de façon systématique l'influence d'un ensemble de paramètres sur une variable (par exemple, le temps de réalisation d'une tâche donnée). Bien sûr, comme ces paramètres sont très nombreux, il n'est pas toujours possible ou raisonnable d'en étudier toutes les combinaisons. Heureusement, il existe des stratégies qui permettent de restreindre

le nombre des expériences (autrement dit, le nombre de combinaisons de paramètres), sans pour autant nuire à la fiabilité du modèle. Ces stratégies s'appellent des *plans d'expérience* [Jain 91]. Certains outils logiciels existent déjà afin d'automatiser la réalisation de plans d'expérience, et en particulier la construction des modèles à partir de l'analyse statistique des résultats numériques obtenus lors de chaque expérience [Anderson 94].

Pour automatiser complètement la démarche de modélisation, il faut donc permettre à ces outils (i) de décrire les expériences qui doivent être lancées et (ii) de récupérer les données numériques produites par chacune de ces expériences. Implicitement, ce dernier point implique que les phases de construction et de contrôle de la réalisation de chaque expérience doivent être assurées de façon automatique. En ce qui nous concerne, ces expériences se déroulent sur les différents nœuds d'une architecture répartie, certaines expériences pouvant exiger la participation simultanée de plusieurs nœuds.

LoadBuilder est une plate-forme d'expérimentation répartie dont l'objectif est de répondre à chacun de ces besoins.

4.3.1 Contraintes

Il existe déjà une pléthore d'environnements qui permettent de construire des applications réparties, tels que PVM [Geist 94] ou les implémentations de MPI [MPI Forum 94]. Toutefois, les motivations qui ont conduit à l'élaboration de ces environnements sont très différentes des nôtres. Par exemple, nous n'avons pas besoin de schémas de communication complexes, ni de tolérance au pannes, ni même de facilités pour le déboguage réparti.

Au contraire, certaines des qualités que nous recherchons pour notre environnement sont plutôt contradictoires avec celles des environnements répartis déjà existants : plutôt que la robustesse et la richesse fonctionnelle, nous cherchons avant tout la sobriété de l'environnement afin de minimiser son intrusion sur les systèmes étudiés.

Limiter le phénomène d'intrusion est à l'évidence la première contrainte que doit respecter un outil de mesure. Cela implique donc que notre environnement doit être aussi léger que possible, afin de ne pas perturber le comportement des systèmes étudiés. En ce qui concerne le protocole de communication, cela nous a conduit à utiliser le protocole UDP/IP, certes non-fiable, mais qui nous permet de maîtriser parfaitement le volume des messages échangés. Dans notre cas, la non fiabilité de ce protocole ne pose pas de réelle difficulté, dans la mesure où (i) les uniques schémas de communication utilisés en cours d'expérimentation sont de type requête/réponse et (ii) la longueur de chaque requête ou réponse est suffisamment courte pour n'exiger que la transmission d'un unique message.

Bien sûr, le contrôle des expériences aurait pu être obtenu à partir des simples interpréteurs de commandes existants (`sh, tcsh, ksh, ...`) et de commandes d'exécution à distance, telles que `rsh`. Les commandes d'exécution à distance ont été rejetées car elles s'appuient sur le protocole TCP/IP, dont nous ne maîtrisons pas les flots de communication. Quant aux interpréteurs de commande existants, ils ne sont pas prévus pour la gestion de multiples processus s'exécutant simultanément sur plusieurs machines. La mise en place et

surtout le contrôle de plans d'expérience à l'aide de ces interpréteurs de commande devient donc rapidement ingérable lorsque plusieurs machines différentes sont impliquées dans chacune des expériences.

4.3.2 Organisation générale

L'environnement *LoadBuilder* est un ensemble de programmes qui s'organisent selon une architecture à deux niveaux (figure 4.3) : les programmes du premier niveau sont chargés de la gestion répartie de l'environnement et du contrôle des programmes du second niveau, qui sont chargés, eux, de la réalisation des expériences.

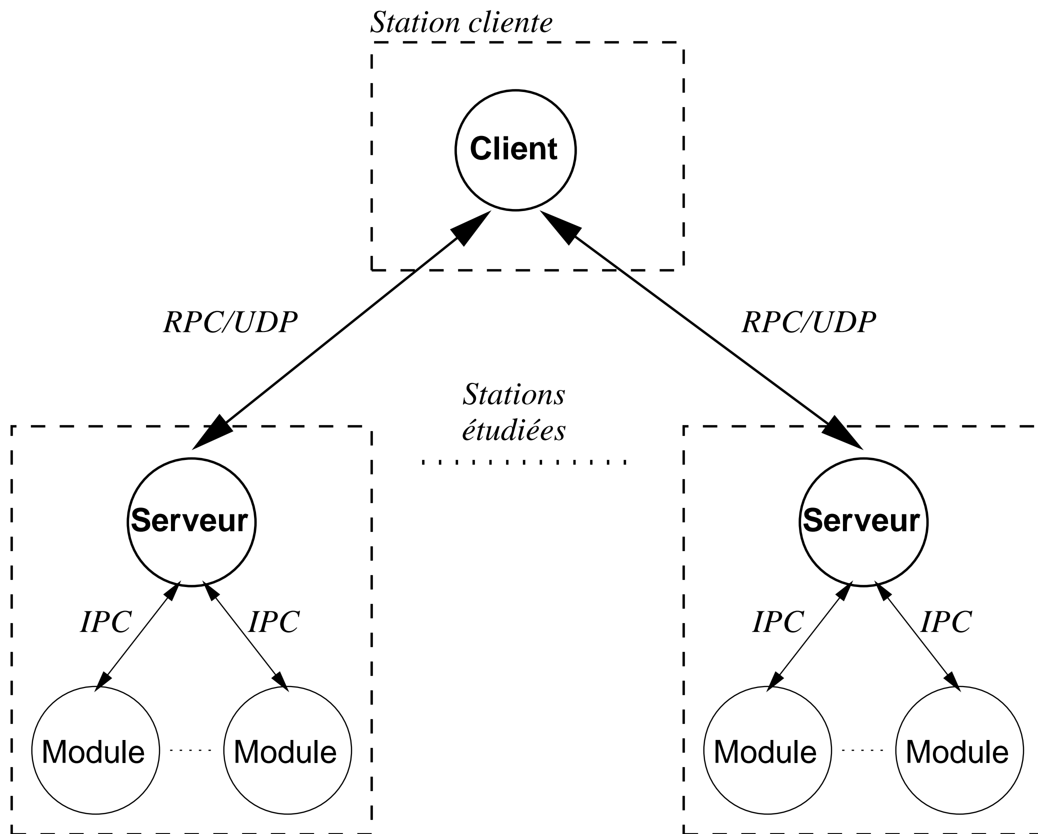


FIG. 4.3: *Architecture de l'environnement LoadBuilder*

Le premier niveau ne comporte que deux programmes : un client, qui s'exécute sur une machine située en dehors de la configuration étudiée, et un serveur, qui s'exécute sur chacune des machines étudiées. Le client interprète les commandes de l'utilisateur et transmet les requêtes correspondantes aux différents serveurs. La communication est réalisée au travers d'un protocole d'invocation de procédure à distance, de type RPC (*Remote Procedure Call* [Sun Microsystems, Inc. 90]). Cette architecture client/serveur est construite au dessus du protocole UDP/IP, afin d'assurer un fonctionnement en mode non connecté. Ce mode de

fonctionnement est intéressant car il nous assure qu'aucune transmission ne peut être déclenchée en dehors des phases d'envois de requêtes et de réception des réponses.

Le second niveau comporte plusieurs petits programmes, que nous appelons des "modules". Ces modules peuvent être classés dans trois catégories, correspondant aux trois types de services proposés par *LoadBuilder* :

- les **modules de charge**, dont la fonction est d'établir les niveaux de charge synthétique désirés ;
- les **modules d'observation**, qui sont chargés de la collecte des informations concernant le fonctionnement et l'activité du système durant les expériences ;
- les **modules de mesure**, qui sont chargés d'évaluer les variations de performance du système en fonction des niveaux de charge établis.

Ces programmes sont lancés et contrôlés par les serveurs sur chacune des machines étudiées. Plusieurs instances de chacun de ces programmes peuvent être lancées pour construire une expérience. Les sorties de type message produites par ces programmes (aux travers de leurs descripteurs de sortie standard et de sortie d'erreur) sont interceptées par les serveurs, qui les conservent jusqu'à ce que le client demande leur récupération (normalement, à la fin de chaque expérience ou série d'expériences). Les sorties de type données, plus volumineuses, sont sauvegardées dans des fichiers temporaires, sous forme binaire. Cette sauvegarde peut s'opérer régulièrement, durant l'expérience ou en une seule fois, à la fin de l'expérience (selon le choix de l'utilisateur). Les serveurs rapatrient ces données à la demande, en assurant la conversion des formats binaires à la volée.

Dans les paragraphes qui suivent, nous décrivons et discutons chacun de ces programmes : le serveur est présenté au paragraphe 4.3.3, le client au paragraphe 4.3.4 (page 82), les modules de charge au paragraphe 4.3.5 (page 83), les modules d'observation au paragraphe 4.3.6 (page 87) et les modules de mesure au paragraphe 4.3.7 (page 89).

4.3.3 Le serveur

Un processus serveur est lancé sur chacune des stations étudiées. Il est chargé d'une part de traiter les requêtes *RPC* reçues du client et, d'autre part, d'assurer la gestion et le contrôle des processus lancés au cours de chacune des expériences.

La majorité de ces processus correspondent à des modules de service de l'environnement *LoadBuilder*. Ces modules, répertoriés à l'annexe A dans le tableau A.2 (page 230) sont utilisés pour construire des niveaux de charge synthétique, effectuer des mesures de performance et collecter les statistiques de fonctionnement du système. Chaque module de service de l'environnement est exécuté par un processus différent.

Les processus restants sont des processus temporaires, utilisés, par exemple, afin de rapatrier les données collectées et stockées durant l'expérience sur chacune des machines. Ces

processus ne sont supposés s'exécuter qu'à la fin de chaque expérience, afin d'éviter toute perturbation dans les mesures.

La mise en place et le contrôle d'une expérience passe généralement par les étapes suivantes :

1. Lancement des modules de collecte de statistiques ;
2. Lancement des modules de charge synthétique ;
3. Lancement des modules de mesure de performance ;
4. Attente de la terminaison des modules de mesure de performance ;
5. Interruption des modules de charge synthétique et de collecte de statistique ;
6. Récupération des données collectées ;
7. Retour du système à l'état initial.

Ce scénario suppose implicitement que les modules, selon leur type, sont conçus soit pour fournir une quantité de travail prédéterminée (modules de mesure et certains modules de charge) soit pour s'exécuter indéfiniment, jusqu'à ce qu'ils soient interrompus (modules de charge et de collecte).

Les serveurs doivent donc proposer les mécanismes adéquats pour supporter ces deux modes de fonctionnement. Pour cela, nous avons mis en place au niveau des serveurs les mécanismes suivants :

– Conservation de l'état des modules : à tout moment, les serveurs peuvent être interrogés afin de connaître le statut d'un ou plusieurs modules (processus en cours d'exécution ou terminé) ;
– Synchronisation avec un ou plusieurs modules : le client peut demander au serveur de l'informer dès qu'il constate la terminaison d'un ou plusieurs modules choisis ;

Ce dernier point pose toutefois le problème de la désignation des modules en cours d'exécution. Afin d'autoriser le maximum de flexibilité, nous avons pour cela décidé de proposer plusieurs mécanismes de désignation des modules :

– désignation directe d'un module, soit à partir de son numéro de processus sur la machine (`pid`), soit à partir de son numéro de séquence[3] ;
– désignation d'un ou plusieurs modules selon leur type ;
– désignation de tous les modules connus par le serveur ;

3. Le numéro de séquence d'un module correspond à son numéro d'ordre dans la séquence de lancement des modules : le $n^{ième}$ module à être lancé porte le numéro de séquence n.

Notons que ces mécanismes de désignation ne sont pas seulement applicables dans le cas des requêtes de synchronisation. Ils sont en fait proposés de façon générale, pour chacune des requêtes exigeant la désignation d'un ou plusieurs modules.

La conservation de l'état des modules au niveau des serveurs pose néanmoins un problème de persistance : il n'est pas utile ni même raisonnable d'exiger que les serveurs conservent indéfiniment l'état de tous les modules qu'ils ont eu à gérer. L'utilisateur a donc la possibilité d'adresser au serveur une requête de type effacement, qui le force à oublier l'état d'un ou plusieurs modules.

En résumé, les requêtes actuellement reconnues par les serveurs sont les suivantes :

- lancer un module de service ;
- retourner le statut d'un module de service (processus en cours d'exécution ou terminé) ;
- terminer l'exécution de modules de service en préservant leurs statut ;
- détruire le statut de modules de services (en terminant au besoin leur exécution si ce n'est déjà fait) ;
- attendre la terminaison d'un ou plusieurs modules de service ;
- rapatrier les données collectées par les modules de service.

4.3.4 Le client

Le client est un interpréteur de commande, qui peut être utilisé de façon interactive ou pour du traitement par lot. Ce client est supposé s'exécuter sur une machine qui ne fait pas partie de la configuration en cours d'évaluation. À l'inverse des serveurs, sa mise en œuvre n'a donc fait l'objet d'aucune précaution particulière pour minimiser le phénomène d'intrusion.

Le mode d'emploi de ce client est donné en annexe A.1 (page 228). La figure 4.4 donne l'exemple d'une session interactive typique, au cours de laquelle l'utilisateur lance une expérience sur trois stations de travail (`cheers, sante` et `prosit`) depuis une quatrième station.

Cet exemple reflète la séquence d'opérations typique d'un schéma d'expérience :

– lancement des processus chargés de la collecte des statistiques système et réseau (`lstat` et `nstat`) ;

– lancement des processus de charge synthétique, comme `lcpu`, qui exécute une boucle infinie afin de charger le processeur ;

- lancement des processus de mesure de performance : dans cet exemple, nous lançons le test de performance du réseau pour le protocole UDP/IP. Ce test, de type "ping-pong" [Burgevin 89, Desprez 90, Dalle 93], mesure le temps d'aller-retour d'un message UDP entre deux machines (ce qui exige l'utilisation de deux processus : `uping` et `upong`) ;

- attente de la terminaison de la mesure : lorsque la durée de l'expérience est variable, comme c'est le cas avec le test "ping-pong", on demande au client d'attendre la terminaison du processus `uping`. Dans le cas contraire, on peut demander au client de se bloquer pour une durée fixée, avant de poursuivre ;

- interruption des services : sans paramètre, la commande `term` s'applique à tous les processus en activité. Elle signale à chacun des processus que l'expérience est terminée et que les résultats peuvent être sauvegardés. Dans cet exemple, cette commande s'adresse implicitement aux processus `lstat` et `nstat`, afin qu'ils enregistrent dans un fichier les données qu'ils ont collectées ;

- récupération des données : dans cet exemple, seules les informations écrites par les processus sur leur sorties standard sont récupérées, au travers de la commande `get`. Les données collectées (statistiques, temps mesurés), qui sont conservées localement dans un fichier, peuvent être transférées vers le client à l'aide d'une autre commande, la commande `coll` ;

- ré-initialisation des serveurs : la commande `kill` tue les processus encore en activité, et demande aux serveurs de supprimer les informations d'état concernant les processus qui ont participé à l'expérience.

4.3.5 Modules de charge

Dans le paragraphe qui suit, nous commençons par discuter les raisons qui nous ont conduit à adopter cette approche basée sur des modules de charge synthétique. Puis nous présentons les quatres types de module de charge proposés : le module de charge du processeur, le module de charge mémoire, les modules de charge réseau et le module de charge système.

4.3.5.1 Discussion

Notre objectif avec les modules de charge, est de parvenir à reproduire les niveaux de charge habituellement observés dans les systèmes que l'on cherche à modéliser.

Dans [Calzarossa 95], les auteurs proposent une classification hiérarchique des types de charge intéressante. Au niveau le plus haut de cette classification, les auteurs placent les applications exécutées par un système. Au niveau intermédiaire, ils font remarquer que l'on peut toujours décomposer ces applications en séquences d'algorithmes. Et au niveau le plus bas, ils poursuivent leur analyse en expliquant que ces algorithmes peuvent à leur tour être décomposés en séquences de routines élémentaires.

Session	Commentaire
`$ LB -I`	*L'option '-I' demande le mode d'exécution interactif*

```
         ***** This is LoadBuilder Rel 0.9 *****

  LoadBuilder is a distributed synthetic workload manager.

Try 'help' or '?' for online manual.
```

Session	Commentaire
LB>**start** cheers sante lstat *#1 <cheers:5172>(running) LSTAT* *#2 <sante:25270>(running) LSTAT*	*Lancement d'un module d'observation local sur* `cheers` *et* `sante`*. En retour, le client indique l'identifiant global du processus, le nom de la machine, le* `pid` *et le statut.*
LB>**start** prosit nstat *#3 <prosit:8423>(running) NSTAT*	*Lancement d'un module d'observation réseau sur* `prosit`*.*
LB>**start** cheers lcpu *#4 <cheers:5173>(running) LCPU*	*Lancement d'un module de charge du processeur sur* `cheers`*.*
LB>**start** cheers upong -o sante *#5 <cheers:5180>(running) UPONG*	*Lancement d'un module secondaire de mesure UDP/IP.*
LB>**start** sante uping -o cheers *#6 <sante:25283>(running) UPING*	*Lancement d'un module principal de mesure UDP/IP.*
LB>**wait** uping *Waiting for 1 services* *Notification from sante: Ok*	*Attente de la terminaison de la mesure UDP.* ... *La terminaison attendue se produit.*
LB>**term** lstat nstat lcpu *#1 <cheers:5172>(running) LSTAT* *#2 <sante:25270>(running) LSTAT* *#3 <prosit:8423>(running) NSTAT* *#4 <cheers:5173>(running) LCPU*	*Terminaison des modules encore actifs.* *En retour, le client affiche la liste des modules concernés.*
LB>**get** lstat nstat uping *#1 LSTAT:'258 lstat.cheers.5172'* *#2 LSTAT:'257 lstat.sante.25270'* *#3 NSTAT:'250 nstat.prosit.8423'* *#6 UPING:'100 uping.sante.25283'*	*Récupérations des messages produits par les modules de type* `lstat, nstat` *et* `uping`*.* *L'affichage indique un nom de fichier et son nombre d'enregistrement. Ces fichiers contiennent les données collectées lors de l'expérience.*
LB>**kill** *#1 <cheers:5172>(sig: 15) LSTAT* *#2 <sante:25270>(sig: 15) LSTAT* *#3 <prosit:8423>(sig: 15) NSTAT* *#4 <cheers:5173>(sig: 15) LCPU* *#5 <cheers:5180>(end: 0) UPONG* *#6 <sante:25283>(end: 0) UPING*	*Terminaison forcée (par défaut) de tous les modules connus, et suppression de leur entrée dans les tables de processus du client et des serveurs.*
LB>**quit**	*Fin de la session*

FIG. 4.4: *Exemple de session interactive avec l'environnement LoadBuilder*

En utilisant cette classification, la charge d'un système peut à tout moment être considérée soit comme la combinaison de l'exécution d'un ensemble d'applications, soit comme la combinaison d'un ensemble d'algorithmes, soit comme la combinaison d'un ensemble de routines élémentaires.

Par conséquent, pour reproduire les niveaux de charge observés sur un système, trois approches sont envisageables : soit construire les niveaux de charge à partir d'applications existantes (compilateurs, commandes UNIX, processeurs de texte, transferts de fichiers, etc), soit construire les niveaux de charge à partir d'algorithmes usuels (produit de matrices, tris, recherche dans des fichiers, etc), soit les construire à partir de routines élémentaires (boucles, accès disque, allocations mémoire, appels système, etc).

Plus la charge est simulée à un niveau élevé dans cette hiérarchie, plus elle est réaliste, c'est-à-dire plus elle correspond à un état de charge réellement observé. À l'inverse, plus la charge est simulée à un niveau bas dans la hiérarchie, plus il est facile d'en contrôler le niveau et la nature.

Notre objectif est de construire un modèle du comportement général d'une machine face à la charge, c'est-à-dire de modéliser les performances de la machine non pas pour certaines situations de charge typiques, mais pour toutes les situations de charge dans lesquelles la machine est susceptible de se trouver. Pour cela, nous avons donc choisi l'approche de bas niveau, qui consiste à combiner l'exécution de routines élémentaires.

Dans les paragraphes qui suivent, nous décrivons successivement chacune de ces routines et les programmes qui les exécutent dans *LoadBuilder*.

4.3.5.2 Charge processeur

Ne charger que le processeur, indépendamment des autres ressources est une tâche facile. Il suffit pour cela d'un processus exécutant une boucle infinie. Une simple boucle ne permet toutefois pas de contrôler finement la charge du processeur, c'est-à-dire son taux d'occupation. En effet, lorsque l'on observe l'occupation d'un processeur dans des conditions réelles d'utilisation, celle-ci oscille entre le niveau d'activité nul et le niveau d'activité permanente.

Pour reproduire (de façon très simplifiée) ces niveaux d'activité intermédiaires du processeur, des phases d'inactivité ont été introduites, au cours desquelles le processus du module de charge du processeur est endormi. Ce processus passe alternativement d'une phase à l'autre. Les durées des deux phases sont fixées, en micro-secondes, au travers de deux paramètres. Le passage de la phase active à la phase inactive est obtenu au travers d'un signal d'horloge programmé. Lorsque la durée de chacune des deux phases est raisonnable (c'est-à-dire de l'ordre de quelques dizaines de milli-secondes[4]), le taux d'occupation du processeur généré par ce module est de l'ordre du ratio de la durée de la période d'activité sur la somme des deux durées.

4. La reprise de l'exécution à la suite d'une phase inactive peut exiger quelques milli-secondes. Expérimentalement, nous avons pu observer (sur une machine Linux cadencée à 200 Mhz) que ce délai devient négligeable lorsque la somme des deux délais atteint 100 milli-secondes et que la durée de chacune des deux phases est au moins de 10 milli-secondes.

4.3.5.3 Charge mémoire

Les architectures récentes de machines s'appuient sur une organisation hiérarchique de la mémoire : la mémoire principale, très lente en comparaison des performances des processeurs, est souvent accédée au travers d'un ou deux niveaux de mémoire cache. De son côté, le système UNIX ajoute un niveau supplémentaire à cette hiérarchie en ayant recours à la pagination afin d'augmenter la capacité mémoire des machines.

Le rôle du module de charge de la mémoire est de solliciter à tous les niveaux de cette hiérarchie, et de façon contrôlée, chacun des mécanismes physiques (cache) et logiciels (pagination) de gestion de la mémoire. Pour solliciter la mémoire cache, il faut que le module accède en permanence à des zones de mémoire distante les une des autres, et toujours différentes. Pour que le mécanisme de pagination se déclenche, il faut que la quantité globale de mémoire manipulée par le processus soit grande.

Ce module sollicite la mémoire en copiant en permance des zones de mémoire. Ces copies sont réalisées entre des adresses de mémoire choisies aléatoirement dans l'espace mémoire du processus. Ce module accepte deux paramètres :

– la taille des zones copiées ;

– la taille de l'espace mémoire.

De la même façon que le module de charge du processeur, le module de charge mémoire alterne des phases d'exécution et des phases d'inactivité.

4.3.5.4 Charge réseau

La charge du réseau se manifeste d'une part au niveau du medium de communication, dont l'utilisation est partagée entre plusieurs machines, et d'autre part, au niveau du système d'exploitation de chaque machine. Au niveau du médium de communication, ce que nous cherchons à modéliser, c'est l'influence de la charge latente du réseau sur les performances de communication d'une machine. Au niveau du système d'exploitation, ce que nous cherchons à modéliser, c'est l'influence du volume de communication généré et/ou subi par une machine sur ses propres performances générales, tant en ce qui concerne ses communications que ses autres activités (notamment l'activité du processeur [Pozzetti 95]).

Pour cela, nous avons défini deux modules de charge, l'un fonctionnant au-dessus du protocole TCP et l'autre au dessus du protocole UDP. Ces deux modules génèrent un trafic uni-directionnel, entre une machine source et une machine destination.

L'intensité et la nature du trafic généré par ces deux modules sont contrôlés au travers des paramètres suivants :

– le délai minimum entre deux envois de messages ;

– le délai maximum entre deux envois de messages ;

- la taille minimum d'un message ;
- le nombre de tailles de message différentes ;
- la différence entre deux tailles de message successives ;

Le délai qui s'écoule entre deux envois de message est choisi de façon alétoire entre le minimum et le maximum. La distribution des choix est uniforme sur l'intervalle. La taille de message effectivement utilisée pour chaque envoi de message est aussi choisie de façon aléatoire (avec une distribution uniforme des choix).

4.3.5.5 Charge système

Le module de charge du système d'exploitation est très similaire au module de charge du processeur. Ce module alterne en effet des phases d'activité et des phases d'inactivité. Durant ses périodes d'activité, ce module exécute des appels système (création et destruction de *sockets*, récupération de statistiques, génération de signaux, création de processus, etc).

4.3.6 Modules d'observation

Les modules d'observation sont chargés de la collecte des informations concernant l'activité de la machine et du ou des réseaux au(x)quel(s) elle est attachée.

Nous proposons deux types de modules d'observation : un module d'observation de type local, chargé de la collecte des informations ne concernant que la machine sur laquelle il s'exécute, et un module d'observation de type global, chargé de la collecte des informations communes à plusieurs machines (en particulier le réseau).

4.3.6.1 Informations locales

Les informations locales sont obtenues auprès du système d'exploitation de chaque machine. La collecte de ces informations est délicate car le nombre et la nature de ces informations varient selon les systèmes d'exploitation.

Par exemple, la répartition des activités du processeur sur la dernière seconde est disponible sur l'ensemble des systèmes UNIX, mais sous des formes variable. Ainsi, Solaris distingue cinq classes d'activités (*idle, user, kernel, iowait, swap*) alors que la majorité des systèmes UNIX n'en distingue que quatre (*idle, user, system, nice*).

Pour faire face à la diversité des informations disponibles, nous définissons les cinq critères de classification suivants :

- **Portabilité**. Les informations récupérées peuvent être soit spécifiques à un système d'exploitation (comme la classe d'activité du processeur *iowait* proposée par Solaris), soit disponibles sur l'ensemble des systèmes d'exploitation (les classes d'activité *idle* ou *user*).

– **Accessibilité**. On peut distinguer trois niveaux d'accessibilité : le *niveau utilisateur*, qui regroupe les sources d'information disponibles pour tous les utilisateurs du système (`uptime`, `who`, `ps`, `...`), le *niveau super-utilisateur*, qui regroupe les sources d'information dont l'accès est limité au super-utilisateur du système (selon la configuration du système, il peut s'agir de commandes telles que `top`, `vmstat` ou `iostat`), et le *niveau noyau* qui regroupe les informations qui sont récupérées directement au cœur du noyau (soit par la consultation d'informations disponibles par défaut, soit par modification/instrumentation du noyau afin de générer de nouvelles informations).

– **Précision**. On peut distinguer deux types de précision, selon l'échantillonnage des informations est réalisé en temps réel ou par des mises à jour périodiques.

– **Nature des informations**. On peut distinguer plusieurs types d'information, comme les compteurs d'évènements, les ratio d'occupation ou d'utilisation des ressources ou encore les quantités brutes de ressources consommées.

– **Spécialisation**. On peut distinguer deux niveaux de spécialisation : le niveau *général*, qui regroupe les indicateurs généraux tels que l'indicateur de charge moyenne des systèmes UNIX, et le niveau *spécialisé* qui regroupe les indicateurs donnant des informations sur certains composants ou sous-systèmes spécifiques, comme le nombre de pages mémoire remisées sur disque ou le nombre de changements de contexte).

Le module de collecte des informations locales actuellement utilisé récupère les statistiques fournies par le service RPC `rstat`. Ce service présente l'avantage d'être largement disponible sur l'ensemble des systèmes UNIX et d'être accessible au niveau utilisateur. Ce service est complet puisqu'il fournit 23 indicateurs, concernant l'activité du processeur, les entrées/sorties, la mémoire, le système d'exploitation et les communications. Il n'offre en revanche qu'une précision relativement faible (la mise à jour des informations a lieu en général une fois par seconde).

4.3.6.2 Informations sur le réseau

La principale vocation de ce module d'observation est de récupérer les statistiques globales concernant l'activité du réseau. En effet, en fonctionnement normal, une machine ne perçoit que le trafic qui la concerne, c'est-à-dire les messages dont elle est l'expéditeur ou le destinataire.

Certains réseaux (Ethernet, en particulier) permettent aux machines de percevoir directement une partie du trafic qui ne leur est pas destiné. En fonctionnement normal, l'interface de ces machines filtre les messages reçus, de telle façon que seuls les messages qui leur sont effectivement destinés soient délivrés au niveau de leur système d'exploitation. Pour que ces machines perçoivent l'intégralité du trafic qui passe par leur interface réseau, il suffit de basculer cette dernière dans un mode de fonctionnement spécifique, appelé mode *promiscuous*[5].

5. Dans les systèmes UNIX, ce mode de fonctionnement est utilisé, par exemple, par des commandes telles que `snoop` ou `tcpdump` [McCanne 93].

Évidemment, ce mode de fonctionnement s'accompagne d'une surcharge conséquente pour la machine, puisque la totalité du trafic perçu par l'interface réseau doit être traité au niveau logiciel, par le système d'exploitation. Sur un réseau Ethernet, cette méthode d'observation reste néanmoins intéressante lorsque les domaines de collision comportent de nombreuses machines, car il suffit de consacrer une machine sur chaque domaine de collision du réseau à cette tâche d'observation. Cette machine sonde peut alors faire office de serveur de statistiques pour l'ensemble des machines du domaine de collision, au travers d'un service RPC par exemple [6].

La tendance actuelle en matière d'organisation des réseaux consiste à réduire la taille des domaines de collision Ethernet (et à en multiplier le nombre), en remplaçant progressivement les répéteurs (*HUB*) par des ponts filtrants (*switch*). La méthode précédente, qui repose sur des machines sondes, devient donc de moins en moins intéressante.

Fort heureusement, la plupart des switch Ethernet récents sont capables d'assurer eux-même cette fonction de sonde. Les statistiques qu'ils collectent en temps réel sur chacun des domaines de collision qu'ils délimitent sont conservées localement dans des bases de données (MIB) dont l'organisation est décrite de façon standard par la syntaxe abstraite du modèle OSI (ASN.1) [Rose 90]. Les bases de données de ces *switchs* sont interrogeables au travers du protocole SNMP [SNMPv2 W.G. 96a, Stallings 93]. La nature des informations détenues dans ces bases de données est relativement variable selon les constructeurs. Plusieurs normes existent pour standardiser ces informations. La plus générale et largement répandue est la norme MIB-II (*Management Information Base*) [SNMPv2 W.G. 96b] qui donne (entre autres) des informations concernant le nombre d'octets et de paquets reçus ou envoyés dans chacun des domaines de collision. La norme RMON (*Remote Network Monitoring*) [Waldbusser 95], qui permet d'obtenir des statistiques détaillées spécifiques au réseaux de type Ethernet (compteurs de paquets, de collision, d'erreurs de parité, de paquets trop petits ou trop gros, etc) est aussi largement implantée sur les *switchs* Ethernet récents.

C'est cette deuxième solution qui a été choisie pour mettre en œuvre le module d'observation du réseau de *LoadBuilder*. Ce module interroge régulièrement la base MIB RMON des *switchs* Ethernet pour en collecter les statistiques.

4.3.7 Modules de mesure

Le rôle des modules de mesure est d'évaluer les performances du système dans ses différentes activités, en fonction des niveaux de charge établis à l'aide des modules de charge. À l'exception des modules de mesure qui concernent le réseau, ces modules sont simplement des versions instrumentées des modules de charge, dont le rôle est d'échantillonner à intervalle de temps régulier le nombre d'opérations réalisées.

La mesure des performances des communications consiste à évaluer le temps d'aller-retour d'un message en fonction de sa taille. En effet, comme les horloges des machines sont

---

6. Les versions anciennes du système d'exploitation des machines Sun (SunOS 4.1.3) permettaient d'ailleurs la mise en place d'un tel service (le service RPC `etherstat`).

désynchronisées, la mesure du temps de réalisation d'une transmission doit être entièrement réalisée par une même machine. Autrement dit, on ne peut pas déclencher un chronomètre sur une machine et l'arrêter à partir d'une autre, car l'action d'arrêter le chronomètre exige la transmission d'un message.

Le module de mesure des performances de communication utilise un algorithme de type "ping-pong" [Burgevin 89, Desprez 90, Dalle 93] : un processus (ping) envoie un message à un autre (pong), qui le lui renvoie immédiatement. Afin d'améliorer la précision de la mesure, le module ne mesure pas le temps d'exécution d'un simple aller-retour, mais le temps d'exécution de plusieurs aller-retours (typiquement, 1000 à 10000 aller-retours). Cet algorithme a été implémenté pour les protocoles TCP et UDP.

Au-dessus d'UDP, la mesure est plus délicate qu'au dessus de TCP, car des messages peuvent se perdre, ce qui impose de mettre en place un mécanisme de détection de perte. Pour cela, nous avons mis en place un délai de garde pour chacun des messages transmis par le processus ping.

Lorsqu'un message est envoyé et que le délai expire sans que le message envoyé ne soit revenu, le message est considéré comme perdu. Cette méthode pose néanmoins deux problèmes : (i) le choix du délai de garde et (ii) l'influence de la perte d'un message sur le résultat de la mesure. En ce qui concerne le choix du délai de garde, nous avons opté pour une solution adaptative dans laquelle le module de mesure découvre automatiquement le délai de garde requis. Nous utilisons pour cela une démarche incrémentale : à chaque fois qu'un message est considéré à tort comme perdu, la mesure est interrompue et reprend avec un délai plus long. Toutefois, pour éviter que la recherche d'un délai satisfaisant ne soit trop longue, il est possible d'indiquer au module un délai initial plus grand que le délai initial par défaut. Pour ce qui est du deuxième problème, les conséquences de la perte d'un message sur le résultat de la mesure sont simplement corrigées en retirant autant fois le délai de garde du temps total de la mesure que de messages ne se sont perdus.

Les deux modules de mesure TCP et UDP sont lancés au travers de deux commandes *LoadBuilder*, l'une pour le processus "ping" et l'autre pour le processus "pong". La commande associée au processus ping contrôle l'expérience (voir algorithme 4.1), au travers des paramètres suivants :

- le nom de la machine sur laquelle s'exécute le processus pong ;
- le numéro de port utilisé par le processus pong ;
- le nom de l'interface réseau locale qui doit être utilisée pour l'expérience ;
- le numéro de port utilisé par le processus ping ;
- la longueur minimale d'un message ;
- la longueur maximale d'un message ;
- le nombre de longueurs de message qui doivent être utilisées entre les deux valeurs précédentes ;

ALG. 4.1 Algorithme de mesure du processus ping

```
début
  calculer_longueurs_messages()
  pour r:=1 jusqu'à nombre de répétitions de l'expérience
  faire
    pour l:=1 jusqu'à nombre de longueurs de messages
      faire
        long←choisir_longueur(l)
        envoyer_longueur(long)
        attendre_réponse()
        heure_début←horloge()
        pour i:=1 jusqu'à nombre d'aller-retours
        faire
          envoyer_message(tampon,long)
          recevoir_message(tampon,long)
        fait
        heure_fin←horloge()
        durée←(heure_début − heure_fin)
        enregistrer(long,durée)
      fait
      dormir(délai)
   fait
fin
```

– le nombre d'aller-retours pour chaque longueur de message ;

– le nombre de répétition de chaque mesure ;

– un indicateur booléen indiquant si les longueurs de message doivent être choisies selon l'ordre croissant des longueurs ou de façon aléatoire ;

– le délai entre chaque répétition de l'expérience ;

– le délai entre chaque longueur de message testée.

De son côté, le processus pong n'accepte que deux paramètres, l'un définissant le numéro de port et l'autre le nom de l'interface réseau utilisée.

4.4 Conclusion

Dans ce chapitre, nous avons présenté nos travaux concernant le problème de la construction d'indicateurs de charge multi-critères et multi-dimensions en environnement hétérogène. Ce type d'indicateur nous semble particulièrement adapté afin d'élaborer des stratégies de répartition dynamique de charge sur les réseaux et grappes de stations de travail actuels. En effet, sur ces architectures, le problème de la répartition dynamique de charge ne se pose plus, en pratique, que pour les applications parallèles et réparties. Or, les caractéristiques remarquables de ces applications sont (i) qu'elles utilisent de façon intensive l'ensemble des ressources disponibles, et (ii) que leur exécution est souvent prise en charge par un environnement d'exécution tel que PVM ou MPI. La première caractéristique implique que les indicateurs doivent prendre en compte l'ensemble des ressources et donc être multi-critères. La seconde implique que les applications sont généralement capables de coopérer avec le mécanisme de répartition de charge en lui fournissant des informations a priori sur leurs tâches. Pour exploiter ces informations de façon efficace, il faut donc que les indicateurs de charge soient multi-dimensions, car cela permet de mettre en correspondance les besoins des tâches avec les disponibilités en ressource des différentes machines.

Nous avons donc commencé par décrire de quelle façon il était possible de mettre en correspondance ces besoins et ces ressources, à partir de l'évaluation d'une fonction de coût. Nous avons ensuite présenté la méthodologie permettant de construire les éléments de cette fonction de coût, au travers d'une démarche de modélisation empirique. Puis, nous avons présenté la plate-forme d'expérimentation répartie *LoadBuilder*. Nous avons conçu et implémenté cette plate-forme répartie dans le but de faciliter et d'automatiser la réalisation des nombreuses expériences exigées par cette démarche de modélisation empirique.

L'application de cette méthodologie représente un travail de modélisation et de mise au point très conséquent. La plate-forme d'expérimentation étant opérationnelle, ce travail de modélisation a déjà pu commencer, notamment avec la modélisation des performances des communications sur un réseau de travail. Parallèlement à ce travail de modélisation, nous avons participé aux travaux de conception de DLB [Tanzy 97], une plate-forme répartie d'aide à la répartition dynamique de charge. Cette plate-forme fonctionne suivant le modèle

client/serveur, au travers d'un service RPC s'exécutant sur chaque site. Ce service se charge de la mise en œuvre des politiques d'information et de localisation, mais laisse aux applications clientes le soin d'assurer elles-même leur politique de transfert. Lorsqu'elles cherchent un site d'exécution pour une nouvelle tâche, les applications qui le désirent contactent le serveur DLB de leur site en lui soumettant une description des besoins de la tâche (ressources, contraintes de placement). Le serveur analyse cette demande et retourne une liste des meilleurs sites d'exécution pour cette tâche.

À terme, notre objectif est donc d'intégrer progressivement les résultats de nos modélisations à la plate-forme DLB, afin d'en améliorer graduellement l'efficacité.

Chapitre 5

Communications multipoints fiables entre systèmes UNIX

5.1 Introduction

L'utilisation des réseaux de stations de travail pour supporter l'exécution d'applications réparties met en évidence certaines lacunes des systèmes UNIX, en particulier dans le domaine des communications.

En effet, si les systèmes UNIX, incontournables dans la majorité des réseaux de stations de travail, disposent de mécanismes suffisants pour répondre aux besoins élémentaires de communication point-à-point des applications réparties, ils sont en revanche relativement démunis en matière de communications multipoints.

Or, les applications réparties ont souvent recours à ce type de communications pour réaliser, par exemple, les opérations suivantes :

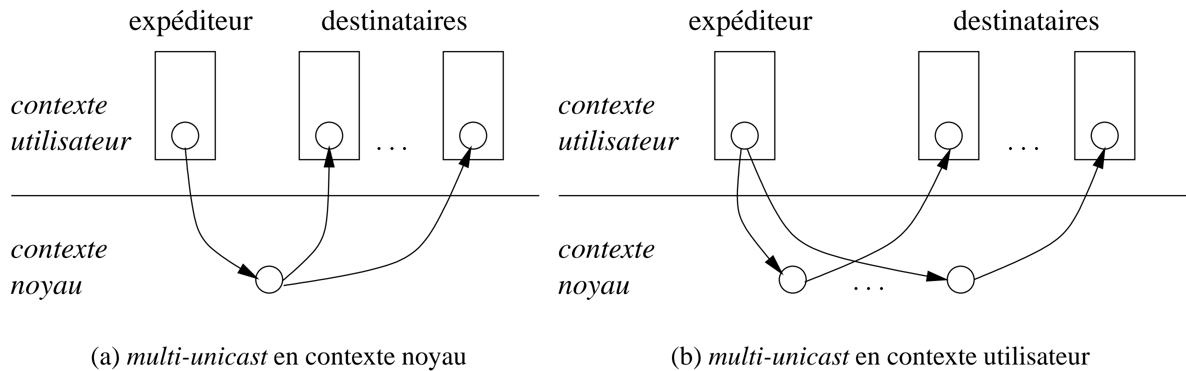
- synchronisation de leurs tâches, au travers de “barrières de synchronisation” ;
- diffusion d'une même information depuis une tâche vers plusieurs autres tâches, au travers de schémas de type “un-vers-plusieurs” (“*one-to-many*”, *multicast*) ou de type “un-vers-tous” (“*one-to-all*”, diffusion/*broadcast*) ;
- rassemblement des informations disséminées de plusieurs tâches, au travers de schémas de type “plusieurs-vers-un” (“*many-to-one*”, rassemblement/*gather*) ;
- redistribution des informations entre leurs tâches, au travers de schémas de type “plusieurs-vers-plusieurs” ou “tous-vers-tous” (“*many-to-many*”, “all-to-all”, échange total/*gossip*)

Ces schémas de communication exigent généralement la transmission *fiable* des messages au sein de *groupes de communication*. Une transmission est dite fiable dès lors que (i) en l'absence de pannes matérielles, toutes les données transmises sont reçues par leur(s) destinataire(s) et (ii) l'intégrité des données transmises est préservée. Un groupe de communication est une abstraction désignant de façon logique, généralement au travers d'un identifiant unique, un ensemble de processus que l'on désire impliquer dans une même communication.

Bien sûr, ces schémas peuvent toujours être obtenus à partir de schémas de communication plus simples, tels que ceux actuellement proposés par la suite de protocoles TCP/IP : une diffusion fiable, par exemple, peut s'obtenir au moyen des communications point-à-point fiables du protocole TCP, en envoyant successivement le même message à chacun de ses destinataires (on parlera alors de *multi-unicast*). Cette diffusion peut aussi être obtenu en ajoutant des accusés de réception, positifs ou négatifs, au protocole de diffusion multipoints non-fiable *IP/Multicast* (lorsque ce dernier est disponible).

L'approche consistant à laisser les applications construire les schémas de communication multipoints dont elles ont besoin à partir de ces communications élémentaires comporte néanmoins des inconvénients.

Au niveau des performances, tout d'abord, cette approche est rarement optimale. La technique du *multi-unicast*, par exemple, s'avère coûteuse non seulement en temps de réalisation,



(a) *multi-unicast* en contexte noyau

(b) *multi-unicast* en contexte utilisateur

FIG. 5.1: *Niveau de mise en œuvre d'un schéma de type* multi-unicast

mais aussi en ressources consommées, lorsqu'elle est réalisée en contexte d'exécution utilisateur plutôt que directement en contexte d'exécution du noyau. Le temps de réalisation est plus long car le message doit être copié dans le contexte du noyau à chaque fois qu'il est envoyé vers un destinataire différent (figure 5.1-b). Outre l'inutilité de ces multiples recopies en mémoire du message, cette façon de procéder implique de nombreux changements de contexte (chaque écriture se faisant au travers d'un appel système).

Ensuite, cette méthode est coûteuse en ressources, car (i) le surplus de travail demandé diminue d'autant la disponibilité du processeur, et (ii) elle peut entraîner, tant pour le système que pour le processus utilisateur, l'allocation massive et redondante de ressources, comme les descripteurs de fichiers (un vers chaque processus destinataire, lorsque le protocole utilisé est TCP).

Enfin, l'application n'est pas seulement pénalisée par de faibles performances, mais elle doit en plus prendre systématiquement en charge un certain nombre de fonctionnalités élémentaires, tant en ce qui concerne la réalisation des communications, qu'en ce qui concerne la gestion des groupes de communication.

Or la complexité de mise en œuvre de ces fonctionnalités peut s'avérer d'autant plus grande que les contraintes sémantiques fixées par l'application peuvent être fortes [Liang 90] : certaines applications, telles que la répartition de charge, où des machines demandent ou proposent de la charge à certaines autres machines, peuvent se satisfaire de transmissions relativement peu fiables et désordonnées, mais rapides, alors que d'autres, telles que les serveurs de fichiers répliqués, peuvent exiger un mécanisme de diffusion qui soit avant tout atomique et totalement ordonné.

La démarche qui s'impose naturellement pour résoudre un tel problème, consiste à proposer un mécanisme générique, qui permette de décharger les applications du détail de ces fonctionnalités élémentaires, tout en leur garantissant un niveau de performance satisfaisant. Comme nous l'avons vu au chapitre 3, bon nombre de travaux ont été menés autour de ce problème, tant au niveau le plus bas des protocoles de communication, qu'au niveau des systèmes d'exploitation ou des boîtes à outils et environnements de programmation. Certaines de ces solutions, telles que PVM [Geist 94], Isis [Birman 86] ou Chorus [Rozier 88], sont

largement utilisées, car elles répondent aux besoins spécifiques de différentes communautés. Cependant, aucune d'entre elles ne permet, à notre connaissance, de répondre de façon globale aux besoins de la majorité des applications réparties (c'est-à-dire toutes communautés confondues) susceptibles d'exploiter les capacités de calcul d'un réseau de stations de travail.

La solution que nous proposons, MPCFS (*MultiPoints Communications File System*), démontre que la possibilité existe de construire un mécanisme de communication multipoints pour les systèmes UNIX, qui permette de satisfaire un maximum de ces besoins, tout en minimisant les défauts observés dans chacune des approches existantes.

Avant de décrire plus en détail ce mécanisme, nous allons commencer par expliciter de façon plus précise le cahier des charges que nous nous sommes fixé.

5.1.1 Plate-forme visée

Implicitement, les caractéristiques de la plate-forme visée sont les suivantes :

- **dimension** : la plate-forme peut comporter jusqu'à plusieurs centaines de machines ;
- **type de machine** : les nœuds de calcul sont des stations de travail ou ordinateurs personnels (PC) a priori hétérogènes ;
- **système d'exploitation** : chaque machine est supposée fonctionner en environnement UNIX et communiquer, en point-à-point, au travers de la suite de protocoles IP ;
- **réseau** : le réseau d'interconnexion, dont la topologie est connue, est formé d'un ou plusieurs réseaux locaux, n'ayant pas nécessairement la capacité de diffusion ; certaines machines peuvent éventuellement être attachées à plusieurs de ces réseaux [1] ;
- **niveau de concurrence** : chaque machine est multi-tâches et multi-utilisateurs et donc a priori multi-applications ;

5.1.2 Qualités souhaitées

Idéalement, le mécanisme proposé devrait avoir les qualités suivantes :

- **performant** : il doit restituer autant que faire se peut les performances du réseau de communication sous-jacent (latence et bande passante) ;
- **efficace** : il doit éviter de gaspiller inutilement les ressources du système et du réseau ;
- **multi-utilisateurs et multi-tâches** : ce mécanisme doit pouvoir être utilisé simultanément par chacun des processus du système ;

1. Il est courant dans les grappes de stations de trouver deux interfaces réseau par machine : l'une traditionnelle (Ethernet, par exemple) assurant la connexion des machines de la grappe avec le reste du réseau local et permettant l'accès aux services habituels via IP (NFS, NIS, DNS, ...), et l'autre plus performante (FastEthernet, Myrinet, SCI, ATM, ...), réservée aux communications au sein de la grappe et ne supportant pas nécessairement le protocole IP.

- **équitable** : la concurrence entre les différents processus qui y ont simultanément recours doit être gérée de façon à assurer à chacun d'eux une qualité de service comparable ;
- **portable** : il doit pouvoir être adapté à la majorité des systèmes UNIX existants ;
- **transparent** : son utilisation ne doit pas impliquer de dépendances particulière vis-à-vis d'un système ou d'un type de réseau du point de vue de la programmation ;
- **robuste** : son fonctionnement doit rester cohérent en cas de panne (par exemple lors du redémarrage d'une machine) ;
- **fonctionnel** : il doit permettre la réalisation de tous les schémas de communication multipoints requis par les applications ;
- **convivial** : son API doit être aussi simple que possible ; il doit si possible proposer des fonctionnalités permettant de faciliter le déboguage et la mise au point des applications ;
- **évolutif** : il doit permettre le support de nouvelles technologies réseau et de nouveaux protocoles ;
- **sécurisé** : les modalités de son utilisation, de même que la spécification des ensembles de machines autorisées à coopérer doivent être paramétrables ;
- **confidentiel** : il doit proposer des fonctionnalités permettant à une application (resp. un utilisateur) d'autoriser ou d'interdire la lecture de ses messages par d'autres applications (resp. utilisateurs) à son insu.

5.1.3 Choix de l'interface de programmation

La mise en œuvre de mécanismes de communication peut s'opérer à deux niveaux : soit (i) au niveau du système d'exploitation (Amoeba, Totem/STREAMS[Rao 96]), soit (ii) au niveau utilisateur, au travers de bibliothèques et d'environnements de programmation (Isis/Horus, Transis, PVM, . . .).

Choix du contexte d'exécution. L'une ou l'autre solution ont leurs avantages, mais ces derniers sont généralement antagonistes. Par exemple, proposer ces mécanismes au niveau utilisateur offre une grande liberté de conception de l'API, tout en permettant assez facilement de s'abstraire des spécificités d'un système d'exploitation donné, et donc d'assurer la portabilité des applications. Les proposer au niveau système permet en revanche d'envisager des solutions beaucoup plus efficaces et performantes, mais en restreignant d'autant plus la portabilité de la solution choisie que sa dépendance vis-à-vis d'un système ou d'une famille de systèmes est forte.

Les deux approches extrêmes pour illustrer ce propos, sont d'un coté les environnements tels que PVM [Geist 94], et de l'autre les systèmes répartis tels que Amoeba [Mullender 90]

ou Chorus [Rozier 88]. PVM assure la portabilité des applications qui l'utilisent du simple fait que tout utilisateur peut, de lui-même, installer cet environnement sur la grande majorité des systèmes existants. En contrepartie, ses performances sont loin d'être optimales, car il ne peut exploiter que les mécanismes de communication qui sont mis à sa disposition en contexte utilisateur par le système d'exploitation.

A l'opposé, une application qui a recours aux fonctionnalités performantes mais spécifiques des systèmes Amoeba ou Chorus, ne pourra que difficilement être portée sur d'autres systèmes.

Il existe bien sûr de nombreuses possibilités de compromis entre ces deux extrêmes. Certains constructeurs ont, par exemple, proposé des versions optimisées de PVM pour leur système (PVMe sur IBM SP2 [Bernaschi 95], par exemple). Mais ces versions ont généralement dû être enrichies de nouvelles fonctionnalités afin d'exploiter au mieux les performances de leur(s) machine(s). En conséquence, si ces ajouts ne remettent pas en cause la portabilité des applications développées antérieurement à leur apparition, ces dernières ne sauraient bénéficier "gratuitement" des optimisations apportées, c'est-à-dire sans modification du code source. Qui plus est, en multipliant les mécanismes de communication, une telle approche incrémentale ajoute encore à la complexité de conception et d'optimisation des applications réparties, en exigeant de leur concepteur une expertise toujours plus grande.

En ce qui concerne les systèmes répartis, nombre d'entre eux proposent une compatibilité POSIX [POSIX.1 90], qui leur ouvre les portes du monde UNIX. Cette ouverture n'est toutefois qu'à sens unique, les applications développées pour ces systèmes compatibles UNIX perdant justement leur portabilité dès lors qu'elles font appel à des fonctionnalités spécifiques, non supportées par les systèmes UNIX traditionnels. Mais la principale difficulté posée par l'utilisation des systèmes répartis est en fait d'ordre sociologique [Jia 96] : les utilisateurs et plus encore, les responsables des moyens informatiques, sont généralement peu enclins à franchir le pas pour adopter un système dont les qualités sont manifestes, mais ne bénéficiant pas d'un support commercial et industriel suffisant pour leur inspirer confiance. Au contraire, comme l'expérience le prouve, un système de médiocre qualité, mais bénéficiant d'un support commercial conséquent peut parvenir à s'imposer comme un standard.

Contraintes. En résumé, proposer de nouvelles fonctionnalités de communication qui ne remettent pas en question la portabilité des applications qui les utilisent, mais tout en garantissant un niveau de qualité de service élevé implique de satisfaire les contraintes suivantes :

– implanter ces mécanismes au cœur du système d'exploitation, afin d'assurer une qualité de service optimale (au sens large, c'est-à-dire non seulement sur le plan des performances, mais aussi par exemple au niveau du partage des ressources ou de la sécurité) ;

– réutiliser et enrichir des mécanismes qui sont largement disponibles sur les systèmes existants, afin de garantir à terme la portabilité de la solution proposée.

Les solutions envisageables. En ce qui concerne la deuxième contrainte, une solution pourrait être d'enrichir les mécanismes de communication ou, plus généralement, de traitement des flots de données existants. Dans le cas des systèmes UNIX, les deux mécanismes les plus largement répandus sont les *sockets*, d'une part, et le mécanisme des *STREAMS*, d'autre part [Comer 93, Padovano 93, Stevens 90].

Construire une solution à partir de ces mécanismes comporte néanmoins des inconvénients :

- ces deux mécanismes sont initialement conçus pour des communications point-à-point ; leur réutilisation dans un contexte multipoints, plus exigeant tant par le nombre que par la diversité des fonctionnalités qu'il requiert est donc problématique ;
- l'API des *sockets*, historiquement la plus ancienne, est certes largement répandue ; mais elle souffre de nombreux défauts de conception[2] qui sur un plan pratique, la rendent déjà très difficile à utiliser avec des communications point-à-point ; ajouter encore à cette complexité ne paraît donc pas raisonnable.
- l'API des *STREAMS*, apparue après celle des *sockets* est mieux conçue que cette dernière. Elle est en particulier très modulaire et évolutive, ce qui est une qualité indéniable. Mais la disponibilité de ce mécanisme, qui est issu des évolutions d'UNIX dans la branche "Système V", n'est en revanche pas garantie sur tous les systèmes UNIX.

Nous nous sommes donc intéressés à une troisième possibilité, encore peu explorée dans ce contexte : le mécanisme de *VFS* (*Virtual File System*) [Kleiman 86]. Ce mécanisme, de plus en plus répandu dans les systèmes UNIX récents [Vahalia 96, Card 93], permet de redéfinir ou de surcharger tout ou partie des opérations concernant les fichiers (et répertoires) d'un système de fichiers UNIX (figure 5.2).

La solution du système de fichiers virtuel. Le *VFS* est une interface générique pour les systèmes de fichiers. Elle a été initialement conçue pour permettre au système de supporter de façon transparente les différents types de systèmes de fichiers existants, et lui permettre d'évoluer facilement vers de nouveaux types. Le *VFS* réalise donc l'interface entre l'unique système de fichiers logique qui est proposé aux utilisateurs, et les multiples systèmes de fichiers physiques sous-jacents, dont l'organisation et les modalités d'accès varient selon le type.

Le `procfs`[3] [Faulkner 91] est un exemple intéressant des possibilités offertes par ce mécanisme. Sous ses formes les plus simples, il permet d'accéder aux données concernant les processus (comme celles que produit la commande `ps`). Sous ses formes les plus élaborées (telle que celle proposée par Linux [Beck 98], par exemple), il permet aux processus d'accéder à tout ou partie des statistiques de fonctionnement du système d'exploitation, en les présentant qui plus est de façon "humainement" compréhensible (format ASCII). Ainsi, dans

2. Le choix de ses concepteurs d'introduire de nouveaux appels systèmes et un nouveau paradigme, et donc de s'écarter de la philosophie initiale des systèmes UNIX est par exemple très discutable.

3. Traditionnellement accessible par le chemin `/proc` sur les systèmes UNIX.

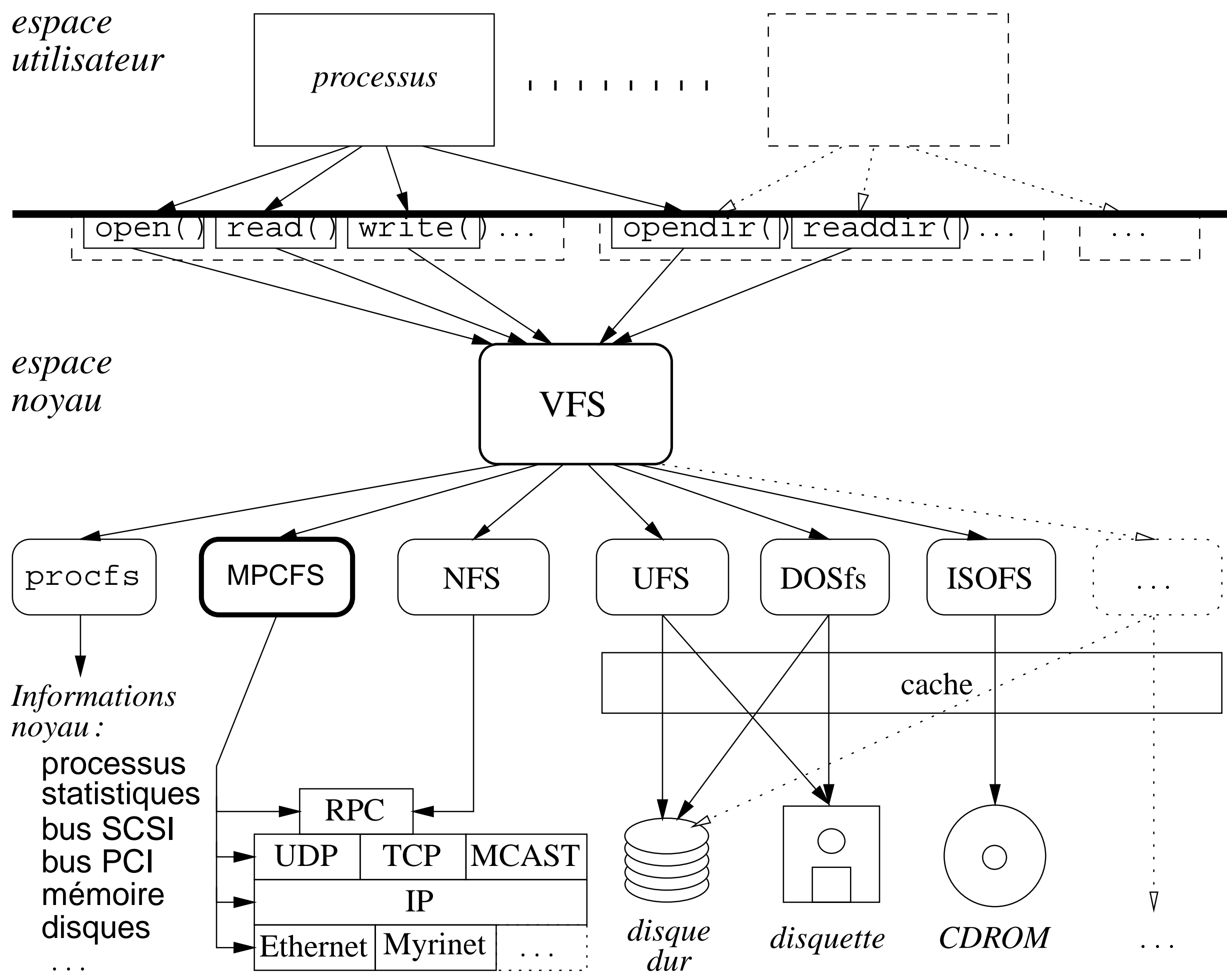


FIG. 5.2: *Le mécanisme de VFS*

ce cas précis du procfs, cette solution offre une souplesse incomparable, tant d'utilisation que de contrôle, par rapport à la solution classique consistant à aller chercher les informations directement dans la mémoire du noyau du système [Graham 95]. En effet, la lecture des informations dans le noyau est une opération complexe, exigeant généralement l'utilisation de fonctions d'interface spécialisées (telles que kvm_open(), kvm_read(), ...). D'autre part, ouvrir l'accès, ne serait-ce qu'en lecture, à la mémoire du noyau (au travers d'un unique fichier, généralement /dev/kmem) est potentiellement dangereux pour la sécurité et la confidentialité du système. Les systèmes qui ne disposent pas du procfs sont donc généralement obligés de proposer des commandes spécialisées dans la récupération de telle ou telle information (ps, iostat, vmstat, ...), auxquelles on a autorisé l'accès en lecture au fichier /dev/kmem.

Le déplacement des fonctionnalités depuis les processus exécutant ces commandes vers les fichiers d'un système de fichiers tel que procfs, est intéressant à plus d'un titre :

– sur le plan des performances, le lancement d'un processus est une opération beaucoup plus coûteuse que la simple consultation d'un fichier ;

– sur le plan pratique, la récupération des informations auprès d'un processus ou directement auprès du noyau, est beaucoup moins aisée qu'à partir d'un fichier ;

– sur le plan de la sécurité, le jeu des permissions permet à un système de fichiers de limiter l'accès aux différentes informations proposées de façon très sélective et lisible ;

5.2 Présentation de MPCFS

Dans l'absolu, MPCFS peut prétendre appartenir à la famille des systèmes de fichiers répartis [Levy 90, Vahalia 96, Tanenbaum 95], à laquelle appartiennent les systèmes de fichiers *NFS* (*Network File System*, de Sun Microsystems) [Sandberg 85, Sun Microsystems, Inc. 89, Callaghan 95], *AFS* (*Andrew File System*, développé à l'Université de Carnegie Mellon) [Satyanarayanan 85], *RFS* (*Remote File System*, de AT&T) [Rifkin 86] ou *DFS* (*DEcorum File System*, conçu par Transarc pour l'environnement DCE) [Kazar 90].

En pratique, il s'agit toutefois d'un parent éloigné, car à la différence de ces autres systèmes de fichiers répartis, dans MPCFS l'objectif n'est pas d'établir un lien entre un processus local et un ou plusieurs fichiers distants mais plutôt d'établir une communication entre un ou plusieurs processus locaux et un ou plusieurs processus distants, indépendamment de tout support physique. Mais de façon sous-jacente, le principe reste le même : lorsqu'un processus écrit dans l'un des fichiers de communication du système du fichiers MPCFS, une communication est déclenchée vers la ou les machines cibles afin d'envoyer des données, et réciproquement, une lecture permet de lire des données transmises depuis une machine distante.

Le système de fichiers RFS, dont l'un des objectifs prioritaires est de respecter à la lettre la sémantique des systèmes de fichiers UNIX, se trouve de fait obligé de fournir un tel mécanisme, pour être en mesure de supporter les fichiers de type "tube nommés" (*named pipes*).

Bien entendu, conformément à la sémantique des fichiers UNIX, ces tubes nommés n'autorisent que des communications point-à-point. Mais comme nous allons le voir plus en détail par la suite, le mécanisme que nous proposons avec MPCFS pour la réalisation de communications multipoints n'est pas, en pratique, très éloigné du principe de fonctionnement des tubes nommés, dont nous nous sommes d'ailleurs inspirés.

Les différences qui existent entre MPCFS et les autres systèmes de fichiers répartis, ne se limitent pas à ce seul "détournement" sémantique des fonctions de lecture et d'écriture. MPCFS exploite aussi pleinement les possibilités de surcharge offertes par le mécanisme de *VFS*, pour permettre la réalisation d'autres opérations que la transmission des données écrites ou lues par ses utilisateurs. En ce sens, MPCFS fonctionne un peu à la manière d'un fichier spécial de périphériques, ou plutôt d'une hiérarchie de fichiers spéciaux, puisqu'ils sont organisés de façon arborescente. Nous nous sommes pour cela largement inspirés du principe de fonctionnement du système de fichiers `procfs` de Linux, dont nous avons discuté les qualités au paragraphe précédent.

5.2.1 Mode d'utilisation

MPCFS est un système de fichiers UNIX, c'est-à-dire une structure arborescente, formée de répertoires et de fichiers d'apparence et d'utilisation conformes aux systèmes de fichiers UNIX traditionnels : chaque fichier et répertoire possède un propriétaire, un groupe propriétaire et des permissions qui établissent les droits en lecture, écriture et exécution accordés à chaque utilisateur.

L'arborescence est composée de plusieurs types de fichiers et répertoires, ayant chacun un rôle particulier (que nous décrivons au paragraphe 5.3). Cette arborescence évolue dynamiquement, selon les opérations que réalisent les utilisateurs sur ces fichiers et répertoires.

Ces évolutions de l'arborescence ne sont toutefois pas toujours obtenues de façon directe, par les opérations habituelles. Les utilisateurs n'ont par exemple pas le droit de créer de fichiers dans l'arborescence ; les seuls fichiers présents sont soit des fichiers inamovibles (comme les fichiers permettant d'envoyer des messages), soit des fichiers apparus de manière spontanée (les messages émis ou reçus). Les utilisateurs ont la possibilité de créer des répertoires dans certaines parties de l'arborescence, mais tous les répertoires n'apparaissent pas nécessairement à la suite d'une opération de création, à l'aide de la commande `mkdir`. Certains répertoires peuvent en effet apparaître de façon indirecte, à la suite d'une opération d'écriture sur un fichier de l'arborescence.

Pour être utilisable, MPCFS doit, comme tout système de fichiers, être intégré au reste de l'arborescence de la machine par une opération de montage. Ce montage, qui intervient généralement automatiquement lors de l'initialisation de la machine, permet de spécifier le chemin à partir duquel débute l'arborescence MPCFS. Bien que ce ne soit pas une obligation, nous supposerons toujours par la suite que ce chemin est `/mpc`.

Le schéma de communication élémentaire proposé par MPCFS est la diffusion. Le ou les utilisateurs définissent l'ensemble des processus concernés au moyen de *groupes de communication* (dont les caractéristiques sont présentées ci-après, au paragraphe 5.2.2).

Ces communications sont réalisées de la même façon qu'avec les "tubes nommés" UNIX, par de simples opérations de lecture ou d'écriture sur certains des fichiers de l'arborescence MPCFS. Ces fichiers, que nous appelons des *nœuds de communication*, portent toujours le même nom dans le système de fichiers : `chan`. En revanche, ils peuvent aparaitre en plusieurs endroits de l'arborescence.

L'envoi d'un message s'obtient donc en écrivant son contenu dans l'un des fichiers `chan` de l'arborescence, matérialisant l'extrémité du tube. A destination, le message peut être lu de façon symétrique en lisant les données disponibles dans l'un des fichiers `chan` de l'arborescence MPCFS de la machine cible (éventuellement la même que celle utilisé pour l'ecriture), correspondant à une autre extrémité du tube.

Ainsi, l'envoi (diffusion) et la réception d'un message peuvent par exemple s'écrire de la façon suivante en langage C (le rôle des répertoires traversés sur le chemin permettant d'atteindre le fichier `chan` est décrit plus en détail au paragraphe 5.3) :

```
/* envoi du message */
int fd=open("/mpc/grp/self/chan",O_WRONLY);
write(fd,message,taille_message);

/* réception du message */
int fd=open("/mpc/grp/self/chan",O_RDONLY);
read(fd,message,taille_message);
```

L'arborescence MPCFS d'une machine peut comporter un grand nombre de fichiers `chan`. Toutefois, pour des raisons que nous expliquons plus en détail par la suite (au paragraphe 5.3), la position de ces nœuds de communication dans l'arborescence (autrement dit, le chemin par lequel ils sont atteints) est très importante : un message transmis par l'intermédiaire du nœud de communication d'un répertoire donné ne peut être consommé que par l'intermédiaire d'un nœud de communication placé au même endroit dans *l'arborescence image* des processus receveurs.

D'une façon générale, tous les mécanismes de communication et d'administration mis en œuvre dans le système de fichiers MPCFS sont accessibles par des opérations de manipulation de fichiers ou de répertoires des systèmes UNIX : ouverture, lecture, écriture de fichiers ; création, destruction, imbrication ou consultation de répertoires.

Ce mode de fonctionnement présente de nombreux avantages. Tout d'abord, il signifie qu'une application distribuée écrite pour le système de fichiers MPCFS ne nécessite pas l'utilisation (et donc l'apprentissage) de librairies ou de langages de programmation particuliers. Les applications ainsi écrites sont de fait portables et relativement indépendantes de la plateforme initiale de développement. Ensuite, il permet au programmeur d'utiliser sans aucune limitation la très large panoplie des outils UNIX traditionnels de manipulation des fichiers et des répertoires.

L'exemple d'envoi et de réception d'un message donné ci-dessus en langage C, peut ainsi

être réécrit très simplement en langage de commande[4] :

```
# envoi d'un message
echo $message > /mpc/grp/$$/chan

# réception du message
message="`head -1 /mpc/grp/$$/chan`"
```

Notons enfin, que la mise au point d'applications distribuées est non seulement facilitée par la possibilité offerte au programmeur de choisir son langage de programmation, mais elle est aussi facilitée par les nombreuses possibilités qui s'offrent à lui pour mettre au point ses programmes grâce aux outils UNIX classiques. Par exemple, chaque message reçu est conservé sous la forme d'un fichier dans un répertoire de stockage jusqu'à ce qu'il soit consommé par tous ses destinataires. Dans ces conditions, déterminer si un message a bien été reçu ou évaluer le nombre de messages en attente et leur taille respective s'obtient de façon très simple, au moyen de la commande `ls`, par exemple.

5.2.2 Communications multipoints

Dans MPCFS, les communications multipoints sont réalisées de façon classique, au travers de *groupes de communications* [Tanenbaum 95].

Les processus s'enregistrent ou se retirent d'un groupe de communication en écrivant le nom de ce groupe dans un *fichier spécial d'enregistrement*. Ce fichier de l'arborescence MPCFS s'appelle `register`, et est accessible en écriture à tout processus de la machine. Un processus peut au besoin, demander à s'enregistrer dans plusieurs groupes de communication.

A sa création, un groupe de communication n'inclut par défaut que la machine locale. Les processus appartenant à un groupe de communication peuvent ensuite décider d'étendre leur groupe de communication à d'autres machines, en faisant une demande d'association. De la même façon que pour son enregistrement, un processus obtient l'association d'un groupe de sa machine avec celui d'une autre machine en écrivant son nom dans un *fichier spécial d'association*, appelé `activate`.

Par exemple, les trois commandes suivantes permettent au processus qui les exécute de demander à faire partie du groupe `grp` et d'y activer les machines B et C :

```
echo -n "+grp" > /mpc/register
echo -n "B" > /mpc/grp/activate
echo -n "C" > /mpc/grp/activate
```

Notons que le mécanisme d'association que nous venons d'évoquer, et qui est décrit plus précisément au paragraphe 5.3.2.10 (page 125), est symétrique : les groupes de com-

4. La différence au niveau du chemin utilisé pour accéder au fichier `chan` s'explique pour des raisons techniques, sur lesquelles nous reviendrons plus loin, au paragraphe 5.3.2.4, page 118.

munication de deux machines sont associés indifféremment à l'initiative de l'une ou l'autre machine.

5.2.2.1 Routage

Chaque message émis (écrit) dans un groupe de communication est mis localement à disposition de tous les autres processus appartenant à ce groupe, et diffusé auprès de toutes les machines *localement* connues pour faire partie du groupe. Cette diffusion ne concerne que les machines localement connues, car MPCFS ne met en place aucun mécanisme de *routage automatique*, ce qui, comme nous allons le voir, n'est pas nécessairement un défaut. Rappelons que le routage est l'action qui consiste à associer un chemin physique (une route) entre l'expéditeur d'un message et son destinataire logique. Un mécanisme de routage automatique est donc un mécanisme (algorithme et/ou protocole), qui calcule cette route de façon automatique, quelles que soient les localisations physiques de l'expéditeur et du destinataire logique.

Avantages et inconvénients de l'absence de routage. Comme nous l'avons vu au chapitre 3, le routage multipoints est un problème très complexe, pour lequel on peut difficilement imaginer trouver une solution qui soit optimale en toute circonstance [Rumeur 94].

Dans le cas d'un réseau local, par exemple, l'utilisation du protocole IP/Multicast, peut s'avérer catastrophique, car ce protocole a tendance à "inonder" le réseau. Le seul moyen dont disposent les applications ayant recours à IP/Multicast pour limiter la propagation de leur trafic, et donc réduire ce phénomène d'inondation, consiste à donner une valeur raisonnable au champ `TTL` du protocole, qui indique le nombre maximal de routeurs qu'un paquet peut franchir. Hélas, les réseaux locaux n'ont pas toujours recours à des routeurs pour cloisonner leur trafic. Dans le cas d'un réseau local Ethernet, le recours à IP/Multicast est donc particulièrement désastreux car tout le trafic IP/multicast généré en tout point du réseau est diffusé sur chacun des domaines de collision du réseau, avec les conséquences que l'on imagine lorsque le réseau comporte plusieurs centaines, voire même plusieurs milliers de machines.

Clairement, dans le cas d'un réseau local Ethernet, mettre en place une solution qui n'aboutit pas au problème d'inondation que nous venons d'évoquer, exige de prendre en compte la topologie du réseau, en mettant en place des arbres de diffusion et/ou en ayant recours au *multi-unicast* (multiple transmission d'un même message vers des destinataires différents).

En fait, l'absence de mécanisme de routage automatique n'est problématique que lorsque les communications multipoints sont réalisées entre des machines qui ne se connaissent pas a priori (l'un des objectifs de IP/Multicast).

Avec MPCFS, notre objectif est plutôt d'apporter une réponse au problème des communications multipoints à l'échelle d'un réseau local. On peut donc raisonnablement supposer que les processus participant aux communications sur chaque machine connaissent les autres machines susceptibles d'être intéressées par leurs communications. D'autant plus que du fait de la symétrie du mécanisme d'association, cette connaissance n'a pas besoin d'être totale :

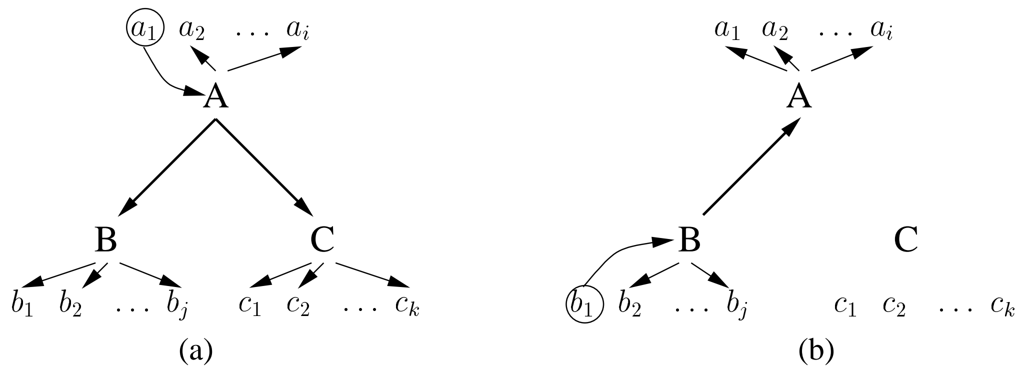
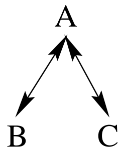


FIG. 5.3: *Exemple de topologie virtuelle de groupes de communication*

il suffit que chaque machine connaisse les autres machines ou, simplement, que chaque machine soit connue des autres machines.

Topologies virtuelles. Le mécanisme d'association et l'absence de routage automatique permettent de créer des topologies virtuelles entre machines : supposons par exemple que le groupe `grp` ait été créé sur trois machines, de noms A, B et C, pour répondre aux besoins d'une application répartie.



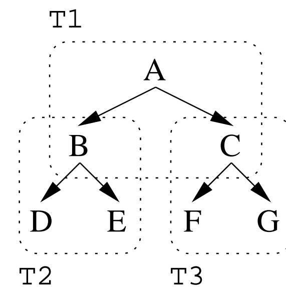
L'absence de routage signifie que les processus de A peuvent choisir d'associer leur groupe avec celui des machines B et C, tout en laissant la possibilité aux processus de B et C de ne pas associer leur groupe respectif (la topologie résultante est présentée ci-contre). Dans une telle situation, les messages émis dans le groupe par des processus s'exécutant sur A, sont diffusés auprès des machines B et C (figure 5.3-a), alors qu'à l'inverse, ceux émis dans le groupe par les processus de la machines B, par exemple, ne sont envoyés qu'à destination de la machine A (figure 5.3-b).

Réalisation du routage au niveau applicatif. L'absence de routage automatique impose donc de déplacer la fonction de routage au niveau applicatif. Comme nous l'avons vu, ceci peut permettre de construire une solution optimale, dès lors que l'application a la connaissance de la topologie du réseau.

Il est néanmoins souhaitable que cette fonction puisse être assurée de façon très simple pour l'application. MPCFS permet pour cela la définition de sous-groupes de communication au sein d'un groupe de communication, au moyen d'un mécanisme de multiplexage hiérarchique (que nous présentons au paragraphe 5.3.2.3, page 116). Ce mécanisme permet en effet de construire plusieurs topologies virtuelles de machines au sein d'un même groupe de communication.

Pour assurer la fonction de routage, l'application doit donc mettre en place ses topologies virtuelles en fonction des schémas de communication qu'elle utilise.

Supposons par exemple qu'une application s'exécutant sur sept machines (A à G) veuille construire l'arbre de diffusion présenté ci-contre. Il lui suffit pour cela de créer trois topologies : la première, T1 reliant la machine A aux machines B et C ; la seconde T2, reliant la machine B aux machines D et E ; et la troisième, T3, reliant la machine C aux machines F et G.



La création d'une telle topologie est relativement simple : il faut que sur chacune des machines, un processus membre du groupe s'affilie à un sous-groupe du groupe de communication portant (par exemple) le nom de la topologie. L'application doit ensuite mettre en place des processus relais sur les machines B et C, afin de retransmettre les messages reçus de A, par la topologie T1, vers les machines D, E, F et G au travers des deux autres topologies. Pour cela, un processus exécutant le programme suivant, en langage de commandes, est suffisant (dans ce cas, il s'agit du processus relais de la machine B) :

```
while [ 1 ]
do
   cat /mpc/grp/$$/T1/chan > /mpc/grp/$$/T2/chan
done
```

5.2.2.2 Primitives d'envoi et de réception

A terme, MPCFS devrait supporter touts les modes de transmission que nous avons présenté au paragraphe 3.2.2 (page 41). Dans l'état actuel d'avancement du prototype développé pour le système Linux (décrit dans la troisième partie de la thèse), les versions bloquantes et non bloquantes des deux primitives sont proposées. Aucune des deux formes de synchronisation présentées n'est en revanche supportée. Le prototype propose pour l'instant une primitive intermédiaire semi-asynchrone : le processus appelant est bloqué jusqu'à ce que son message soit pris en charge par la couche de transport pour chacune de ses destinations.

5.2.2.3 Caractéristiques des groupes de communication

Les caractéristiques principales des groupes de communication proposés par MPCFS sont les suivantes (cf. paragraphe 3.2.1, page 39) :

- **Dynamicité** : Les groupes de communication proposés dans MPCFS sont dynamiques transitoires.

  Dans sa version actuelle, MPCFS ne propose pas encore de mécanisme de notification passif pour signaler l'arrivée ou le départ des processus dans un groupe de communication. La détection active des processus locaux enregistrés dans un groupe est en revanche possible.

  Précisons aussi que dans cette version, aucun moyen n'est proposé afin de permettre aux processus membres d'un groupe sur une machine de connaître les processus

membres de ce groupe sur les autres machines (cette information est difficile et, potentiellement, très coûteuse à maintenir de façon permanente) ; si cette information est indispensable, elle peut toujours être obtenue au niveau applicatif (il suffit qu'un processus établisse la liste des membres sur chaque machine et la diffuse dans le groupe en fonction des besoins de l'application).

- **Modalités d'accès** : Dans MPCFS, les modalités d'accès aux fonctionnalités d'un groupe de communication sont dissociées de la notion d'appartenance au groupe. Elles sont régies directement au travers du mécanisme de contrôle d'accès (permissions) sur les fichiers et répertoires de l'arborescence.

  Pour mettre en place un groupe fermé (selon la terminologie employée par [Tanenbaum 95]), il suffit donc d'interdire l'écriture (et/ou l'accès) sur les fichiers `chan`. À l'inverse pour mettre en place un groupe ouvert, il suffit de permettre l'écriture sur ces fichiers. Les permissions sur les fichiers offrent par ailleurs beaucoup plus de possibilités que la simple autorisation ou interdiction de la fonction d'émission. Elles s'appliquent à toutes les autres fonctions d'un groupe de communication, comme la lecture ou la consommation des messages, la création et destruction de sous-groupes, ou l'association de machines.

  Les règles d'administration de chaque groupe, qui définissent quels processus ont le droit de modifier les permissions précédentes, sont fixées par l'administrateur de la machine. Elles peuvent être ajustées au niveau d'un groupe ou d'un ensemble de groupes (les groupes faisant partie d'un ensemble de groupe sont désignés au moyen d'une expression régulière portant sur le nom logique du groupe).

- **Égalité des processus** : MPCFS propose des groupes égalitaires ; tous les processus membres du groupes peuvent envoyer des messages et chaque message transmis dans le groupe par l'un de ses processus membres est systématiquement délivré à chacun des autres membres du groupe.

- **Gestion répartie** : La gestion des groupes proposée dans MPCFS est totalement répartie.

- **Adressage** : Dans MPCFS, l'adressage est de type logique. Les destinataires des messages sont désignés de façon implicite par le nom du groupe ou du sous-groupe dans lequel l'émission est réalisée (les messages sont mis à disposition de tous les membres présents dans le groupe ou le sous-groupe) ; les groupes et sous-groupes sont identifiés par une chaîne de caractères choisie par l'utilisateur (et donc indépendante de tout protocole sous-jacent).

- **Atomicité** : La diffusion atomique est certes souhaitable, mais pas toujours indispensable. Comme elle est d'autre part très coûteuse à mettre en œuvre, elle n'est pas garantie par MPCFS, qui offre des garanties plus faibles mais généralement suffisantes : le système garantit que soit le message est délivré à tous ses destinataires soit une erreur est signalée à son expéditeur. Dans le cas des modes de transmission asynchrones et semi-asynchrones, le délai entre le retour de la primitive d'envoi et la réception de l'erreur (un signal) peut être assez long.

Notons enfin que le temps de réalisation de la diffusion peut être relativement long, en particulier lorsque le message doit être transmis successivement vers plusieurs destinataires différents (*multi-unicast*). Dans ce cas, la disparition “naturelle” de l’une des destinations (c’est-à-dire lorsque le groupe de communication n’existe plus sur la machine destinataire car tous ses processus membres s’en sont retirés) n’est pas considérée comme une erreur. Par ailleurs, toute nouvelle machine destinataire qui apparaît dans le groupe dans ce laps de temps est ignorée.

– **Séquencement** : MPCFS ne propose pour l’instant qu’un séquencement intra-groupe *FIFO* selon la source. Les autre types de séquencement sont néanmoins prévus, car leur mise en œuvre au niveau de l’application peut s’avérer difficile à mettre en œuvre, et peu performante.

– **Extensibilité** : L’extensibilité mesure la capacité du système à faire coopérer de nombreuses machines. La seule limite à l’extensibilité du système de fichiers MPCFS concerne l’espace mémoire requis pour stocker les messages en attente de lecture et les structures de données associées à chaque groupe de communication. Les messages en attente de lecture sont en effet conservés dans la mémoire du noyau, afin d’être toujours disponibles sans délai. De plus, un nombre important de ces messages peut être conservé dans chaque groupe, afin de permettre au système d’être utilisé de façon asynchrone. En revanche, le coût direct de l’ajout d’une machine supplémentaire dans un groupe est relativement réduit car il n’exige que l’allocation de quelques structures de données de taille raisonnable (quelques dizaines à quelques centaines d’octets).

En reprenant la classification proposée par [Liang 90], les groupes de communication proposés dans MPCFS sont de type non déterministe. Par ce choix, nous avons clairement cherché à proposer une solution qui favorise les performances des communications (en particulier au niveau de la latence), mais qui proposent néanmoins un minimum de fiabilité et de séquencement.

5.3 Communiquer grâce à MPCFS

Nous allons maintenant décrire le fonctionnement pratique et le mode d’utilisation du système de fichiers MPCFS. Au paragraphe 5.3.1, nous présentons les principes généraux de fonctionnement et la terminologie utilisée. Puis, au long de la section 5.3.2, nous faisons le tour de chacun des fichiers et répertoires que l’utilisateur peut trouver et utiliser dans l’arborescence MPCFS. Enfin, au paragraphe 5.3.3, nous abordons plutôt les aspects liés à l’administration et la supervision de MPCFS, avec la description des fichiers de configuration du système.

5.3.1 Fonctionnement général, terminologie

Au travers d’une structure arborescente, MPCFS permet la définition de *groupes de communication*, entités permettant d’associer de multiples processus s’exécutant sur les différentes machines d’un réseau de stations de travail UNIX.

Communications. Les communications sont obtenues grâce aux opérations classiques de lecture et d'écriture sur certains des fichiers de l'arborescence MPCFS, appelés *nœuds de communication*. Le fonctionnement de ces nœuds de communication est donc comparable à celui des "tubes nommés" du système UNIX, à ceci près qu'ils ont de multiples extrémités, que ces extrémités sont disjointes, et qu'elles peuvent être réparties sur plusieurs machines. Chaque processus membre d'un groupe de communication dispose de son propre ensemble de nœuds de communication. Ils matérialisent chacun l'extrémité de l'un des multiples tubes existant dans un groupe de communication. Par la suite, nous appelons ces tubes des *sous-groupes de communication*.

Arborescence. Ces sous-groupes peuvent être organisés de façon hiérarchique. Cette hiérarchie de sous-groupes est matérialisée au travers de *l'arborescence de multiplexage*, dont dispose chaque processus membre. Les processus peuvent y créer et imbriquer à volonté des *répertoires de multiplexage*, au sein desquels sont placés les nœuds de communication (un par répertoire de multiplexage). Chaque processus membre développe sa propre arborescence de répertoires de multiplexage, indépendamment des autres processus. En créant un répertoire de multiplexage, un processus obtient automatiquement un nouveau nœud de communication, lui permettant d'accéder à un nouveau sous-groupe. Pour faire partie d'un même sous-groupe, les processus membres d'un groupe doivent avoir créé le même répertoire de multiplexage dans leur arborescence de multiplexage. Ils communiquent ensuite dans ce sous groupe au travers de leur nœud de communication respectif, dans ce répertoire de leur arborescence.

Stockage. Chaque message transmis au travers d'un nœud de communication est envoyé vers chacune des machines associées pour laquelle le sous-groupe existe. Sur ces machines, le message est mis à la disposition de chacun des processus appartenant au sous-groupe (c'est-à-dire ayant créé le même répertoire de multiplexage que celui contenant le nœud de communication utilisé pour l'envoi du message). Dans l'attente de sa consommation par chacun de ses destinataires, le message est conservé sous la forme d'un fichier indépendant dans un *répertoire de stockage*. Les répertoires de stockage sont organisés de façon arborescente, de la même façon que les répertoires de multiplexage. L'arborescence de stockage est unique, et reproduit l'arborescence de multiplexage du groupe (elle est l'union des arborescences de multiplexage de chacun des membres du groupe). Chacun des répertoires de l'arborescence de stockage est donc associé à l'un des sous-groupes existant dans le groupe, et contient les messages en transit dans ce sous-groupe (soit en attente de lecture, soit en attente d'émission).

Contrôle. Les opérations de création de groupes de communication et d'ajout de nouvelles machines au sein d'un groupe de communication sont laissées à l'initiative des processus. Elles sont, elles aussi, réalisées par des opérations d'écriture sur des fichiers de l'arborescence MPCFS, que l'on appelle respectivement *fichier d'enregistrement* et *fichier d'association*. En revanche, leur acceptation n'est pas automatique. Elle est placée sous le contrôle de l'administrateur de chaque machine, par l'intermédiaire de fichiers de configuration du système. Les règles définies par l'administrateur du système dans ces fichiers permettent

d'ouvrir ou de limiter l'accès aux différents groupes de communication de façon sélective, en fonction de certains utilisateurs et/ou de certaines machines du réseau de stations de travail.

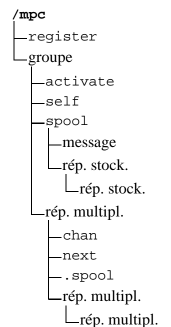
5.3.2 Arborescence MPCFS

Les principaux éléments qui intéressent directement l'utilisateur dans l'arborescence MPCFS sont :

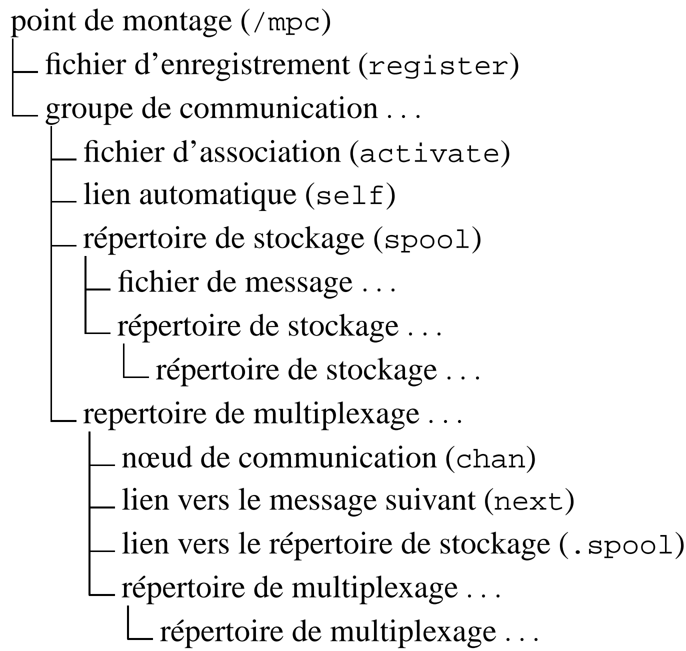
- le *point de montage* du système de fichiers ;
- les répertoires associés aux *groupes de communication* ;
- les *répertoires de multiplexage* ;
- les *nœuds de communication* ;
- les *répertoires de stockage* et *les fichiers contenant les messages* ;
- les fichiers *d'enregistrement* et *d'association* ;
- le *lien automatique* ;
- le *lien vers le message suivant*.

Les paragraphes qui suivent décrivent en détail chacun de ces fichiers et répertoire. La figure 5.4 décrit leur organisation. Un exemple concret d'arborescence est présenté au paragraphe 5.3.4.1, page 128, et un exemple d'utilisation au paragraphe 5.3.4.2, page 130.

5.3.2.1 Point de montage



	Point de montage
NOM	/
TYPE	répertoire
APPARITION	lors du montage du système de fichiers MPCFS (`mount -t mpc`)
DISPARITION	lors du démontage du système de fichiers (`umount -t mpc`)
VISIBILITÉ	tout processus
PARENT	dépendant du point de montage
FILS	• répertoires des groupes de communication (5.3.2.2) • `register` (5.3.2.9) • quelques fichiers réservés, dépendants de l'implémentation
OPÉRATIONS	• lecture (`opendir(),readdir(), closedir()`) • écriture (`chmod(), chown()`) • exécution (`chdir()`)
RÔLE	• racine de l'arborescence MPCFS



Les points de suspension (. . .) indiquent que plusieurs fichiers ou répertoires du même type peuvent apparaitre au même niveau.

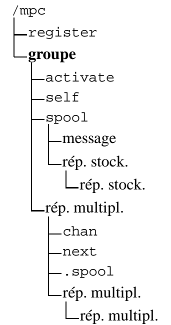
FIG. 5.4: *Schéma d'organisation des fichiers et répertoires de l'arborescence* MPCFS

Ce répertoire apparaît lors du montage du système de fichiers MPCFS, à l'aide de la commande `mount()`. Par exemple, la commande suivante monte le système de fichiers MPCFS sur le répertoire `/mpc` de l'arborescence globale de la machine :

```
root# mount -t mpc - /mpc
```

De façon symétrique, le système de fichiers MPCFS disparaît lors de son "démontage" par la commande `umount()`.

5.3.2.2 Répertoires des groupes de communication



Groupe de communication	
NOM	correspond au nom logique du groupe associé
TYPE	répertoire
APPARITION	lors de l'enregistrement du premier processus membre du groupe
DISPARITION	• lors du dés-enregistrement du dernier processus membre du groupe • lors de la terminaison du dernier processus membre du groupe
VISIBILITÉ	tout processus
PARENT	racine de l'arborescence MPCFS (point de montage)
FILS	• répertoires propres des membres (5.3.2.3) • `spool` (5.3.2.7) • `activate` (5.3.2.10)
OPÉRATIONS	• lecture (`opendir()`,`readdir()`, `closedir()`) • écriture (`chmod()`) • exécution (`chdir()`)
RÔLE	• signaler l'existence d'un groupe communication • permettre l'accès aux fonctionnalités d'un groupe de communication

Les répertoires des groupes de communication apparaissent à la racine du système de fichiers. Ils portent le nom logique donné au groupe.

Ces répertoires ne sont ni créés ni détruits directement par les commandes habituelles de création ou de destruction de répertoires (`mkdir()`, `rmdir`). Le répertoire d'un groupe de communication apparaît lorsque le premier de ses processus membres parvient à s'y enregistrer[5].

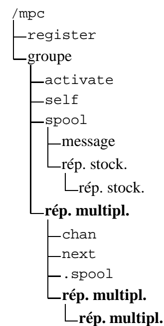
L'utilisateur et le groupe d'utilisateurs propriétaires du répertoire, ainsi que les permissions initialement positionnées sur le répertoire, sont déduites (i) de la configuration établie par l'administrateur du système (voir paragraphe 5.3.3, page 125), et (ii) des indications données lors de la création du groupe par le premier processus à s'enregistrer dans le

5. Un processus s'enregistre ou se dés-enregistre d'un groupe de communication en écrivant le nom du groupe désiré dans le fichier *fichier d'enregistrement*, dont le fonctionnement est décrit au paragraphe 5.3.2.9.

groupe (quand elles ne contredisent pas les premières). Dès lors que le répertoire est créé, ces permissions et propriétés sont ajustables au moyen des commandes classiques (`chmod()`, `chown()`).

A sa création, un groupe de communication n'est associé à aucune autre machine. Les processus membres d'un groupe peuvent étendre ce groupe à d'autres machines, en demandant leur association. (Un processus demande l'association d'une nouvelle machine en écrivant le nom de cette machine dans un fichier, le *fichier d'association*, dont le fonctionnement est décrit au paragraphe 5.3.2.10, page125.)

Le répertoire d'un groupe de communication disparaît de l'arborescence lorsque le dernier de ses processus s'en retire ou termine son exécution. La dissolution d'un groupe est locale : elle ne remet pas en cause l'existence de ce groupe sur les autres machines.



5.3.2.3 Les répertoires de multiplexage

Répertoire de multiplexage *(racine de l'arborescence de multiplexage d'un processus membre)*	
NOM	numéro de processus d'un processus membre
TYPE	répertoire
APPARITION	lors de l'enregistrement du processus membre
DISPARITION	lors du dés-enregistrement du processus membre
VISIBILITÉ	tout processus ayant accès au répertoire du groupe
PARENT	répertoire racine d'un groupe de communication
FILS	• répertoires de multiplexage • `chan` (5.3.2.5) • `next` (5.3.2.6) • `.spool` (5.3.2.7)
OPÉRATIONS	• lecture (`opendir()`,`readdir()`, `closedir()`) • écriture (`mkdir()`, `rmdir()`, `chmod()`) • exécution (`chdir()`)
RÔLE	sous-groupe racine : permet de diffuser vers *tous* les membres du groupe
NOTE	ne peut être ni créé, ni détruit (autrement que par le mécanisme d'enregistrement)

Les répertoires de multiplexage permettent à une application de distinguer plusieurs types de message et donc de gérer virtuellement plusieurs flux de communication indépendants.

Chaque répertoire de multiplexage définit un sous-groupe du groupe communication, c'est-à-dire un sous-ensemble de l'ensemble des processus membres du groupe : les processus membres qui ont créé ce répertoire dans leur arborescence propre font partie du sous-groupe, alors que ceux qui n'ont l'ont pas créé, n'en font pas partie. Les processus créent ou détruisent leurs répertoires de multiplexage de façon traditionnelle, à l'aide par exemple, des commandes UNIX `mkdir` et `rmdir`.

Répertoire de multiplexage *(descendant de la racine de l'arborescence de multiplexage d'un processus membre)*	
NOM	au choix (utilisé pour former le nom logique du sous-groupe associé)
TYPE	répertoire
APPARITION	`mkdir()` dans le répertoire parent
DISPARITION	• lors du dés-enregistrement du processus membre • `rmdir()` dans le répertoire parent
VISIBILITÉ	tout processus ayant accès au répertoire parent
PARENT	répertoire de multiplexage
FILS	• répertoires de multiplexage • `chan` (5.3.2.5) • `next` (5.3.2.6) • `.spool` (5.3.2.7)
OPÉRATIONS	• lecture (`opendir()`,`readdir()`, `closedir()`) • écriture (`mkdir()`, `rmdir()`, `chmod()`) • exécution (`chdir()`)
RÔLE	sous-groupe : permet de diffuser vers *certains* des membres du groupe (ceux qui ont crée ce même répertoire dans leur arborescence)

Les sous-groupes peuvent être imbriqués à volonté, afin de définir des sous-groupes de sous-groupes. Chaque niveau supplémentaire définit à son tour un sous-ensemble des processus membres du niveau précédent. Bien entendu, chaque niveau de l'arborescence peut comporter plusieurs sous-groupes de même niveau, correspondant à des sous-ensembles de membres (éventuellement disjoints) du sous-groupe parent commun.

De la même façon que pour un groupe, un sous-groupe est créé lorsque son premier processus membre crée le répertoire de multiplexage correspondant dans son arborescence propre. Un sous-groupe est détruit lorsque le dernier de ses processus membres détruit le répertoire du multiplexage correspondant au sous-groupe dans son arborescence propre.

Notons que le sous-groupe associé au répertoire est identifié non pas par le seul nom du répertoire de multiplexage, mais par ce nom précédé de la totalité des répertoires de multiplexage traversés depuis la racine de l'arborescence de multiplexage. Les sous-groupes peuvent donc être identifiés logiquement et de façon non ambiguë, en les désignant par leur nom UNIX relativement à la racine de l'arborescence de multiplexage. Ainsi, le répertoire racine porte le nom logique "/", le sous-répertoire `sgrp` de la racine, le nom logique "/sgrp", le sous-répertoire `ssgrp` du répertoire précédent, le nom logique "/sgrp/ssgrp", ...

L'utilité d'un tel mécanisme n'est plus à démontrer. On le trouve par exemple dans l'environnement PVM sous la forme d'une étiquette associée au message (*tag*) et de façon plus élaborée dans la norme MPI, sous la forme de contextes de communication (*communicators*). Typiquement, ce mécanisme est utilisé pour accorder aux messages des niveaux de priorité différents selon leur type, ou encore pour éviter toute interférence entre les messages de diverses composantes d'une application, en particulier lors de la réutilisation de librairies.

Enfin, comme nous l'avons vu au paragraphe 5.2.2.1, ce mécanisme permet implicitement de mettre en place plusieurs topologies virtuelles de communication dans un groupe, chaque sous-groupe regroupant des membres de machines éventuellement différentes.

La racine de l'arborescence de multiplexage est obligatoirement présente dans l'arborescence de multiplexage de chaque processus, car ils ne peuvent pas la détruire. Cette propriété est importante car elle implique que les messages transmis par l'intermédiaire de la racine sont systématiquement adressés à tous les processus membres du groupe de communication.

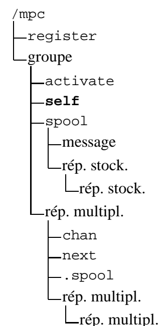
En termes de schémas de communication, la racine de l'arborescence de multiplexage réalise donc la diffusion des messages au sein du groupe de communication, alors que les répertoires de multiplexage permettent la mise en place de schémas de type "multicast" (diffusion entre membres d'un sous-ensemble quelconque des processus appartenant au groupe).

Chaque répertoire de multiplexage contient ou peut contenir, selon le chemin par lequel il est atteint, les éléments suivants :

– le nœud de communication `chan` du répertoire (voir paragraphe 5.3.2.5) ;

– un lien vers le répertoire de stockage qui est associé au répertoire courant nommé `.spool` (voir paragraphe 5.3.2.7) ;

– un lien symbolique vers le premier des messages en attente pour le processus dans le répertoire de stockage : c'est le contenu de ce fichier qui sera récupéré lors de la prochaine lecture du fichier `chan`. Ce message, ainsi que tous les messages présents dans le répertoire de stockage, peut être lu à volonté sans être consommé tant qu'il n'a pas été lu de façon définitive au travers du nœud de communication `chan`. Quand aucun message n'est en attente pour le processus, ce lien n'existe pas (il joue donc aussi le rôle d'indicateur).



5.3.2.4 Le lien automatique

Lien automatique	
NOM	`self`
TYPE	lien symbolique
APPARITION	lors de la création d'un groupe
DISPARITION	lors de la destruction d'un groupe
VISIBILITÉ	processus membres du groupe
PARENT	répertoire d'un groupe de communication
FILS	idem répertoire de multiplexage (5.3.2.3)
OPÉRATIONS	idem répertoire de multiplexage
RÔLE	faciliter l'accès à l'arborescence propre d'un processus
NOTE	ne peut être ni crée, ni détruit (et donc, ni redéfini)

Ce lien symbolique n'est visible que pour les processus membres d'un groupe. Sa destination est fixée dynamiquement en fonction du processus qui l'utilise, vers le répertoire

de multiplexage racine de ce processus (obligatoirement un membre) dans le répertoire du groupe.

Ce lien est donc proposé afin de faciliter à un processus membre l'accès à son arborescence de multiplexage : un processus enregistré dans un groupe de nom “grp” accède simplement à la racine de son arborescence dans ce groupe par le chemin `/mpc/grp/self/`.

D'un point de vue pratique, signalons toutefois que ce lien peut s'avérer difficile, voire impossible à utiliser dans certaines situations. C'est par exemple le cas lorsque le système MPCFS est utilisé au travers de programmes écrits en langage de commande faisant appel à des commandes “externes” (c'est-à-dire exécutées par des processus distincts du processus interprétant les commandes).

Ainsi, dans l'exemple qui suit, la première commande est vouée à l'échec, alors que la seconde peut réussir, si c'est bien le processus qui interprète les commandes qui est enregistré dans le groupe :

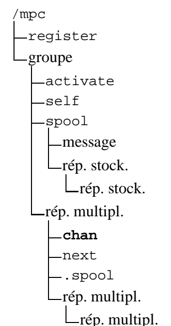
```
$ cat /mpc/grp/self/chan
$ cat < /mpc/grp/self/chan
```

Dans le premier cas, en effet, c'est le processus exécutant la commande `cat` qui essaie d'ouvrir le fichier `/mpc/grp/self/chan`. Or pour ce processus, qui n'est assurément pas enregistré dans le groupe, le lien `self` n'exite pas. Dans le deuxième cas, c'est l'interpreteur de commande lui-même qui ouvre le fichier `/mpc/grp/self/chan`, afin de placer le descripteur de fichier ouvert obtenu, sur l'entrée standard du processus exécutant la commande `cat`.

Dans le cas de programmes écrits en langage de commande, une solution simple et efficace pour éviter la confusion, consiste à systématiquement utiliser la variable prédéfinie donnant le numéro de processus courant (normalement `$$`). Ainsi les deux commandes suivantes sont équivalentes :

```
$ cat /mpc/grp/$$/chan
$ cat < /mpc/grp/$$/chan
```

5.3.2.5 Les nœuds de communication



Les nœuds de communication sont les fichiers qui servent à envoyer et recevoir les messages. Ils portent tous le même nom : `chan`. Chacun des répertoires de l'arborescence de multiplexage contient l'un de ces nœuds de communication. Ils fonctionnent suivant le mode flux de messages.

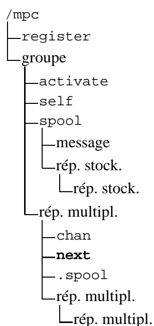
Chaque opération d'écriture sur un fichier `chan` entraîne l'envoi d'un message contenant tout ou partie des données écrites. La taille des messages est en effet limitée[6]. Lorsque la

6. Cette limite dépend de l'implémentation. Dans le prototype actuel, elle est de l'ordre de 10 kilo-octets.

Noeud de communication	
NOM	`chan`
TYPE	fichier
APPARITION	lors de la création du répertoire de multiplexage parent
DISPARITION	lors de la destruction du répertoire de multiplexage parent
VISIBILITÉ	tout processus ayant accès au répertoire de multiplexage parent
PARENT	répertoire de multiplexage (5.3.2.3)
FILS	–
OPÉRATIONS	• lecture (`open(), read(), select(), close()`) • écriture (`open(), write(), select(), close()`) • attributs (`ioctl(), fcntl()`)
RÔLE	envoi et réception de messages dans le sous-groupe associé au répertoire parent
NOTE	• ne peut être ni créé, ni détruit directement • lorsque les permissions en lecture sont supprimées sur ce fichier, les messages reçus dans le sous groupe du répertoire parent sont éliminés

taille des données écrites dépasse la taille maximale d'un message, seul le début du message est envoyé. La quantité d'information effectivement écrites est obtenue en retour de l'appel système `write()`.

De façon symétrique, chaque opération de lecture ne permet que la réception d'un unique message. Un message ne peut être lu, totalement ou partiellement, qu'une seule fois d'un fichier `chan` (comme nous allons le voir au paragraphe suivant, le système propose toutefois un mécanisme complémentaire permettant de lire plusieurs fois un même message). En particulier, si la quantité de données demandée est inférieure à la taille du message, la fin du message est perdue. A l'inverse, si la quantité d'information demandée est supérieure à la taille du message, seule la quantité d'information contenue dans le message est retournée (la valeur de retour de l'appel système `read()` indique la taille effective du message lu).



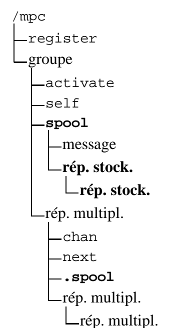
5.3.2.6 Lien vers le message suivant

Ce lien symbolique apparaît dans un répertoire de multiplexage quand un ou plusieurs messages sont disponibles en lecture pour le processus membre dans le sous-groupe associé. Dans ce cas, le lien pointe vers le premier des messages en attente dans le sous-groupe. Ce lien est donc particulièrement utile pour permettre de retrouver le message suivant, et de lire son contenu sans le consommer (équivalent à l'appel de la fonction `recv()` avec le paramètre `MSG_PEEK`, de l'API des *sockets*).

Il permet aussi à un programme écrit en langage de commande de détecter l'arrivée de nouvelles données à lire sans se bloquer en lecture sur un descripteur de fichier (cette fonction qui n'est généralement disponible que dans les langages "avancés", au travers des appels système `select()` ou `ioctl()`).

Lien vers le message suivant	
NOM	`next`
TYPE	lien symbolique
APPARITION	lors de la disponibilité d'un message en lecture dans le sous groupe associé au répertoire parent
DISPARITION	lorsque tous les messages disponibles ont été lu
VISIBILITÉ	tout processus ayant accès au répertoire de multiplexage parent
PARENT	répertoire de multiplexage (5.3.2.3)
FILS	–
OPÉRATIONS	idem fichier de message
RÔLE	• détection de l'arrivée des messages • consultation (sans consommation) du message suivant en attente de lecture
NOTE	ne peut être ni créé, ni détruit directement

5.3.2.7 Les répertoires de stockage



Répertoire de stockage	
NOM	• `spool` lorsqu'il s'agit du repertoire de stockage associé au sous-groupe racine du groupe et qu'il est atteint à partir du répertoire du groupe • `.spool` lorsqu'il est atteint à partir d'un répertoire de multiplexage • le nom du répertoire de multiplexage associé au sous-groupe lorsqu'il est atteint à partir d'un autre répertoire de stockage
TYPE	répertoire
APPARITION	lors de la création du sous-groupe associé
DISPARITION	lors de la destruction du sous-groupe associé
VISIBILITÉ	tout processus ayant accès à l'un des répertoires parent
PARENT	• répertoire du groupe de communication ou répertoire de stockage parent
FILS	• répertoires de stockage fils • messages (5.3.2.8) en transit (en attente de lecture ou d'émission)
OPÉRATIONS	• lecture (`opendir(),readdir(), closedir()`) • exécution (`chdir()`)
RÔLE	stockage des messages en transit

Les répertoires de stockage sont les répertoires contenant les messages en transit dans le groupe, c'est-à-dire soit en attente de lecture, soit en attente d'émission, soit en attente d'un accusé de réception.

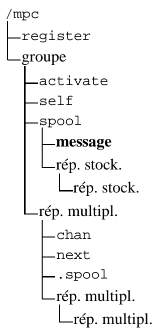
Chaque sous-groupe du groupe de communication possède son propre répertoire de stockage. La capacité de ces répertoires est limitée en nombre de messages[7]. Tant que cette

7. La limite, actuellement dépendante de l'implémentation, est fixée à 512 messages. Cette limite devrait à terme être variable selon les groupes.

limite n'est pas atteinte, les sous-groupes peuvent fonctionner de façon asynchrone. Dès qu'elle est atteinte, tout nouveau message est systématiquement refusé et le fonctionnement du groupe passe en mode synchrone ; il faut alors qu'une émission en cours se termine ou qu'un message soit lu par un processus membre du sous-groupe pour que qu'un nouveau message puisse être émis ou reçu dans le sous-groupe correspondant.

L'arborescence de stockage reproduit l'arborescence des sous-groupes du groupe de communication. Les répertoires de cette arborescence apparaissent ou disparaissent au fur et à mesure des créations ou destructions de sous-groupes dans le groupe.

Bien évidemment, puisqu'un groupe contient toujours au minimum un membre, et que chaque membre d'un groupe est toujours enregistré dans le sous-groupe racine, le répertoire de stockage du sous-groupe racine est toujours présent. Ce répertoire est accessible depuis la racine de l'arborescence de chaque groupe de communication et s'appelle `spool`.



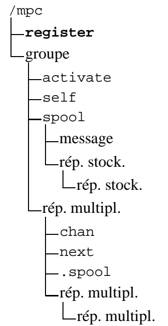
5.3.2.8 Les fichiers contenant les messages

Message	
NOM	un numéro entre 1 et le nombre maximal de messages supportés par un répertoire de stockage
TYPE	fichier
APPARITION	• lors de la réception dans le sous-groupe d'un message venant d'une autre machine • lors de l'envoi par un processus local, d'un message dans le sous-groupe
DISPARITION	• lorsque le message est consommé par tous ses destinataires • lorsque le message a été transmis à la machine destinataire et celle-ci l'a accepté
VISIBILITÉ	tout processus ayant accès au répertoire parent
PARENT	répertoire de stockage (5.3.2.7)
FILS	–
OPÉRATIONS	• lecture (`open()`,`read()`, `close()`)
RÔLE	• permettre la (pré)lecture d'un message • permettre d'évaluer le taux remplissage d'un répertoire de stockage

Les fichiers contenant les messages permettent d'apprécier le remplissage d'un repertoire de stockage (nombre et taille des messages en transit). Ils permettent aussi de lire les messages en transit (en particulier au travers du lien `next` présenté au paragraphe 5.3.2.6).

Les noms des fichiers contenant chacun des messages (un fichier différent pour chaque message) sont des numéros compris entre 1 et le nombre maximum de messages que peut supporter le répertoire. Bien que le système attribue ces numéros cycliquement, des numéros de message consécutifs ne correspondent pas necéssairement à des messages émis ou reçus en séquence.

5.3.2.9 Le fichier d’enregistrement



Fichier d’enregistrement	
NOM	`register`
TYPE	fichier
APPARITION	lors du montage du système de fichiers
DISPARITION	lors du démontage du système de fichiers
VISIBILITÉ	tout processus
PARENT	point de montage
FILS	–
OPÉRATIONS	écriture (`open(), write(), close()`)
RÔLE	permettre l’enregistrement d’un processus dans un groupe de communication

Le fichier d’enregistrement, nommé `register` est placé à la racine de l’arborescence MPCFS. Il permet à un processus de demander à être enregistré auprès d’un groupe ou à être supprimé d’un groupe dans lequel il est enregistré. Il suffit pour cela que le processus candidat écrive le nom du groupe désiré précédé du symbole ’`+`’ en cas d’adhésion ou du symbole ’`-`’ en cas résiliation.

La demande d’adhésion est alors acceptée ou rejetée (dans ce cas l’opération d’écriture échoue en retournant un code d’erreur) en fonction des autorisations d’accès préétablies pour ce groupe par l’administrateur de la machine.

Par exemple, en écrivant la chaîne de caractères ’`+grp`’ dans le fichier `/mpc/register`, un processus demande à s’enregistrer auprès du groupe de nom `grp`. En cas de succès, ce processus se voit attribuer un répertoire propre dans le répertoire du groupe, dont le nom correspond à son numéro de processus (`pid`). Ce répertoire correspond au sous-groupe racine du groupe de communication. Il peut y accéder par le chemin `/mpc/grp/self/`. Tous les messages qui arrivent dans ce sous-groupe sont conservés et mis à sa disposition à partir de cet instant précis.

Si ce processus ne désire pas recevoir les messages transmis dans ce sous-groupe racine, il lui suffit de retirer les permissions de lecture sur le fichier `/mpc/grp/self/chan`. Il peut autoriser ou interdire à d’autres processus (qui peuvent ne pas être membres du groupe) de consommer les messages qui lui sont transmis grâce aux permissions d’accès et de lecture sur les fichiers de son arborescence propre.

De façon symétrique, un processus se dés-enregistre du groupe précédent, en écrivant la chaîne de caractère ’`-grp`’ dans le fichier `/mpc/register`. Cette opération est réalisée automatiquement par le système lorsque le processus se termine.

Lorsqu’un processus quitte un groupe de communication, ses messages en attente de lecture sont détruits.

L’exemple suivant, en langage de commande, permet au processus qui l’exécute de s’enregistrer dans un groupe de communication de nom `grp`, d’y diffuser la chaîne de caractères

“Hello world !” (dans son sous-groupe racine) et de se dés-enregistrer du groupe de communication :

```
echo -n "+grp" > /mpc/register
if [ $? -ne 0 ]
then
   echo "Echec de l'enregistrement !"
   exit 1
else
   echo "Hello World !" > /mpc/grp/self/chan
   echo -n "-grp" > /mpc/register
fi
```

De plus, la demande d’enregistrement accepte un certain nombre de paramètres optionnels (à la suite du nom du groupe demandé, séparés par des espaces) :

– *les permissions d’accès sur le répertoire du groupe* (valide uniquement lorsque le groupe est crée à la suite de la demande d’enregistrement) : de la forme `g=xxx`, `g-xxx` ou `g+xxx`, où `xxx` est la permisson en octal, et où ’`=`’, ’`-`’ et ’`+`’ indiquent respectivement si la permission doit être exactement positionnée, si elle représente un masque des bits à positionner, ou un masque des bits à retirer (par rapport aux permission attribuées par défaut) ;

– *les permissions du fichier d’activation du groupe* (lorsque le groupe est créé à la suite de la demande) : `a[=+-]xxx` ;

– *les permissions d’accès par défaut aux répertoires de multiplexage* du processus dans le groupe : `s[=+-]xxx` ;

– *les permissions par défaut des nœuds de communication* du processus : `n[=+-]xxx` ;

– *les permissions par défauts des fichiers contenant les messages* : `m[=+-]bbb` où `b` vaut `0` ou `1` ;

– *le mode de fonctionnement par défaut des nœuds de communication du processus* : écriture bloquante (`+wb`) ou non bloquante `-wb`, lecture bloquante (`+rb`) ou non bloquante (`-rb`).

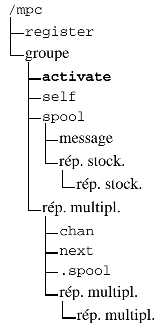
Par exemple, la commande donnée ci-après, signifie que le processus demandeur souhaite s’enregistrer dans le groupe `grp`, dont il désire fermer l’accès aux autres utiliateurs du système (`g-077`). Le paramètre `n-111` indique que le fichier `chan` doit par défaut être créé sans les permissions de lecture (ce qui n’est pas définitif, les permissions peuvent éventuellement être modifiées au travers de `chmod()`). Le paramètre `-wb` indique que les écritures (envois de messages) doivent être par défaut non bloquantes.

```
echo -n "+grp g-077 n-111 -wb" > /mpc/register
```

La demande d’enregistrement échoue si elle ne peut être honorée dans sa totalité, c’est-à-dire (i) si elle va à l’encontre de la configuration établie par défaut par l’administrateur

du système, (ii) si elle est incohérente (par exemple, g-777), (iii) si elle est impossible (permission de type 'g' sur un groupe déjà existant) ou (iv) si elle malformée (par exemple m+777).

5.3.2.10 Le fichier d'association



Fichier d'association	
NOM	activate
TYPE	fichier
APPARITION	lors de la création du groupe
DISPARITION	lors de la destruction du groupe
VISIBILITÉ	tous processus ayant accès au répertoire du groupe
PARENT	répertoire de groupe
FILS	–
OPÉRATIONS	• lecture (open(), read(), close()) • écriture (open(), write(), close())
RÔLE	• permettre d'associer le groupe correspondant au répertoire parent avec un groupe de même nom sur une autre machine • obtenir la liste des machines associées au groupe

Utilisé en lecture, ce fichier permet d'obtenir la liste des machines associées à un groupe de communication. Utilisé en écriture, ce fichier permet à un processus de demander l'association du groupe de communication dans lequel le fichier se trouve avec un groupe de communication portant le même nom, mais sur une autre machine. Il suffit pour cela au processus demandeur d'écrire le nom de la machine dans le fichier.

La demande d'association peut échouer selon différents scénarii, générant chacun un code d'erreur spécifique lors de l'opération d'écriture dans le fichier activate :

1. la demande est refusée car contraire aux règles établies par l'administrateur de la machine locale ;

2. la demande est refusée car contraire aux règles établies par l'administrateur de la machine distante ;

3. la demande est refusée car le groupe n'existe pas sur la machine distante.

5.3.3 Fichiers de configuration

La configuration de MPCFS est réalisée grâce à trois fichiers dont nous allons maintenant décrire le rôle.

5.3.3.1 Configuration initiale

Les paramètres généraux de fonctionnement de MPCFS sont placés dans le fichier `/etc/mpc_rc`, qui n'est lu qu'une seule fois, lors du montage du système de fichiers MPCFS. Ces paramètres, tous facultatifs, permettent de fixer certaines valeurs limites telles que : le nombre maximum de groupes, les limites du groupe en nombre de sous-groupes, de processus membres, de nombre de messages stockés, d'occupation mémoire, etc. Il permet aussi de fixer les valeurs par défaut des permissions et mode de fonctionnement des fichiers et répertoires du système de fichiers (ces valeurs pouvant être modifiées en fonction des groupes, au travers du fichier de configuration des groupes).

5.3.3.2 Définition des groupes

Ce fichier de configuration (par défaut `/etc/mpc_groups`) permet à l'administrateur de la machine de définir les règles de fonctionnement des groupes de communication vis-à-vis de leurs utilisateurs. Il est analysé lorsqu'un processus demande son enregistrement dans un groupe de communication.

L'analyse de ce fichier a plusieurs buts :

1. Vérifier que le processus demandeur a effectivement le droit d'utiliser le nom de groupe qu'il réclame ;
2. Établir la configuration par défaut du groupe lors de sa création :
   - Permissions et propriété du répertoire racine du groupe ;
   - Permissions du fichier d'association (`activate`) ;
   - Permissions des répertoires de multiplexage des membres du groupe ;
   - Permissions des fichiers contenant les messages ;
   - Permissions et mode de fonctionnement des nœuds de communication du processus membre ;
3. Établir les permissions et propriétés que l'utilisateur peut spécifier lors de sa demande d'enregistrement.

Au niveau syntaxique, chacune des lignes est formée de cinq champs (voir annexe B.1, page 238) :

1. La désignation d'un ensemble de noms de groupes (pouvant être défini au moyen d'expressions régulières) ;
2. La désignation d'un ensemble d'utilisateurs ;
3. L'utilisateur et le groupe d'utilisateurs propriétaires du groupe lors de sa création ;
4. La liste des permissions et propriétés par défaut des fichiers et répertoires du groupe ;
5. La liste des modifications que l'utilisateur peut apporter à la configuration par défaut.

Algorithme d'évaluation. Ce fichier, lu lors du montage de la partition MPCFS, est parcouru de façon séquentielle, à chaque fois qu'un processus demande à s'enregistrer dans un groupe (par l'intermédiaire du fichier spécial `register`). La vérification est réalisée, de la façon suivante :

1. Recherche de la *première* ligne applicable : une ligne est applicable dès lors que le nom de groupe demandé fait partie de l'ensemble de noms défini par le premier champ. Si aucune ligne n'est applicable, la demande est rejetée ;

2. Comparaison de l'`uid` du demandeur avec l'ensemble d'utilisateurs défini par le deuxième champ de la ligne : si le demandeur ne fait pas partie de cet ensemble sa demande est rejetée ;

3. Vérification de la validité des paramètres accompagnant la demande et calcul des paramètres effectifs de fonctionnement : si la configuration demandée par l'un de ces paramètres est interdite, la demande est rejetée.

On remarquera en particulier que cet algorithme de vérification implique que l'ordre d'apparition des lignes dans le fichier est significatif car il fixe leurs priorités respectives (décroissante).

5.3.3.3 Le fichier d'export

Ce fichier (par défaut `/etc/mpc_export`) permet à l'administrateur de la machine de définir les règles concernant les associations de groupes locaux avec ceux des autres machines du réseau. Il est consulté lorsque la machine locale et une autre machine désirent s'associer dans un même groupe de communication à la suite d'une demande d'activation (que ce soit à l'initiative d'un processus s'exécutant sur la machine locale ou à celle d'un processus s'exécutant sur la machine distante).

Les règles concernant les associations définissent le type d'interaction autorisée vis-à-vis de la machine distante :

– Aucune : la machine locale refuse l'association ;

– Réception seule : la machine locale ne transmet aucun message à destination de la machine distante (excepté les messages de contrôle) ;

– Émission seule : la machine locale ne peut recevoir aucun message de la machine distante (excepté les messages de contrôle) ;

– Émission et réception : la machine locale autorise à la fois les émissions et réceptions de message.

Bien entendu, lorsque les règles définies localement par chacune des machines sont antagonistes (par exemple, quand les deux machines n'autorisent qu'une association de type émission seule), l'association est finalement refusée.

Chaque ligne significative de ce fichier est formée :

– d'un champ désignant un ensemble de noms de groupes (pouvant être défini au moyen d'expressions régulières) ;

– une succession de champs désignant chacun une machine ou un ensemble de machines (domaine), ainsi que le type d'interaction autorisée vis-à-vis de cette(ces) machine(s).

L'algorithme de vérification mis en place est donc très simple : il parcourt le fichier de façon séquentielle, à la recherche de la première ligne faisant intervenir à la fois le groupe de communication demandé et la machine candidate. Si une telle ligne n'est pas trouvée, l'algorithme refuse l'association.

5.3.4 Exemples

Les deux exemples qui suivent illustrent de façon concrète le fonctionnement et l'utilisation du système de fichiers MPCFS. Le premier présente une arborescence faisant intervenir tous les éléments que nous venons de décrire, et le second donne un exemple d'utilisation en langage "C". En ce qui concerne les fichiers de configuration, des exemples sont donnés en annexe B.

5.3.4.1 Exemple d'arborescence

La figure 5.5 présente un exemple d'arborescence faisant intervenir chacun des élements présentés dans les paragraphes précédents.

– `/mpc` est le point de montage du système (choisi par l'administrateur du système) ;

– `/mpc/register` est le fichier d'enregistrement, qui permet aux processus de s'enregistrer dans un groupe de communication ;

– `/mpc/grp` est le répertoire associé au groupe de communication "grp" (le nom du groupe est choisi par l'application) ;

– `/mpc/grp/activate` est le fichier d'association du groupe "grp", qui permet aux processus d'associer le groupe de communication "grp" de la machine locale à celui d'autres machines ;

– `/mpc/grp/28` est la racine de l'arborescence de multiplexage propre d'un membre du groupe "grp", dont le `pid` est `28`. Ce répertoire est attribué automatiquement à ce processus lorsqu'il s'enregistre dans le groupe ;

– `/mpc/grp/self` représente le lien automatique (vers `28` sur la figure). Il permet aux processus membres d'un groupe d'accéder à leur répertoire propre par un chemin qui ne dépend pas de leur numéro de processus (la destination de ce lien change en fonction du processus qui l'utilise) ;

– `/mpc/grp/spool` représente la racine de l'arborescence de stockage du groupe, dans laquelle sont conservés les messages en transit ;

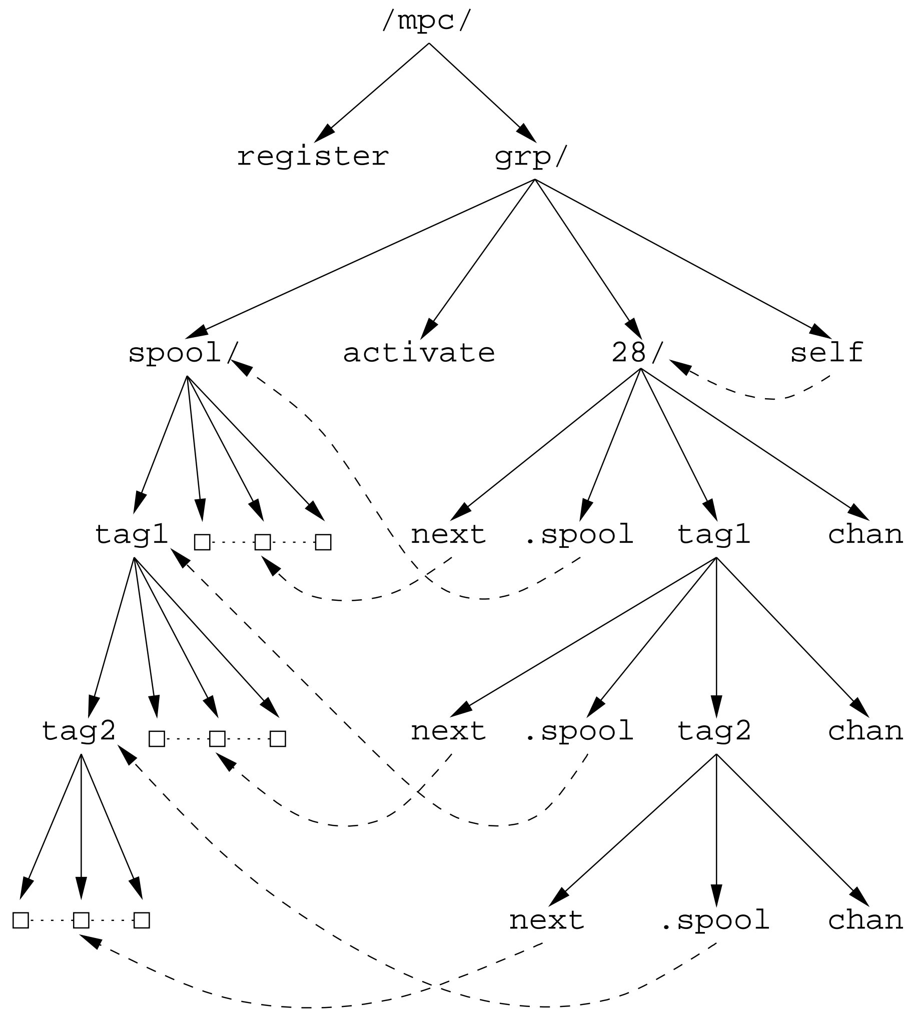


FIG. 5.5: *Exemple d'arborescence MPCFS*

– `/mpc/grp/28`, `/mpc/grp/28/tag1` et `/mpc/grp/28/tag1/tag2`, sont les répertoires de multiplexage du processus 28 qui correspondent aux sous-groupes "/", "/tag1" et "/tag1/tag2" ;

– `/mpc/grp/28/chan`, `/mpc/grp/28/tag1/chan` et `/mpc/grp/28/tag1/-tag2/chan` sont les nœuds de communication du processus 28 permettant d'envoyer et recevoir des messages dans les sous-groupes "/", "/tag1" et "/tag1/tag2" ;

– `/mpc/grp/28/next`, `/mpc/grp/28/tag1/next` et `/mpc/grp/28/tag1/-tag2/next` représentent des liens vers les messages suivants du processus 28 dans les sous-groupes "/", "/tag1" et "/tag1/tag2" ;

– `/mpc/grp/spool`, `/mpc/grp/spool/tag1` et `/mpc/grp/spool/tag1/tag2` sont les répertoires de stockage des sous groupes "/", "/tag1" et "/tag1/tag2" ;

– `/mpc/grp/28/.spool`, `/mpc/grp/28/tag1/.spool` et `/mpc/grp/28/tag1/-tag2/.spool` représentent des liens vers les répertoires de staockage des sous-groupes "/", "/tag1" et "/tag1/tag2" ;

– les boîtes (□) représentent des fichiers contenant des messages.

5.3.4.2 Exemple d'utilisation

L'exemple d'utilisation que nous présentons ici, réalise à intervalle de temps régulier une opération de réduction impliquant un nombre de machines quelconque.

Rappelons que l'objectif d'une opération de réduction est d'obtenir sur une machine, le résultat d'un calcul portant sur les données détenues de façon locale, par chacune des machines (dans notre exemple, il s'agit du calcul du mimimum de la charge locale de chaque machine). Afin de répartir le calcul, une opération de réduction suppose que le résultat du calcul peut être obtenu par l'application successive d'un opérateur binaire associatif (dans notre exemple la fonction $\min(a, b)$).

Le principe de l'opération de réduction consiste donc à répartir le calcul en construisant une arborescence de tâche (une par machine) : chaque tâche qui est placée sur un nœud de l'arbre applique successivement l'opérateur binaire entre sa valeur courante et la valeur reçue de chacun de ses fils dans l'arbre[8], puis transmet le résultat de son calcul vers son père. La tâche destinataire du calcul global est donc placée à la racine de l'arbre (figure 5.6).

Dans l'exemple que nous proposons, les tâches exécutent toutes le même algorithme (paradigme SPMD). Le principe de l'algorithme est le suivant : chaque tâche (processus) reçoit le nom de son père en paramètre, à l'exception de la racine, qui sait alors qu'elle est le destinataire du calcul. Chaque tâche s'enregistre dans un groupe de nom "un_groupe" et reçoit les valeurs transmises de ses éventuels fils au travers d'un sous-groupe dont le nom correspond à son nom de machine. A intervalle de temps régulier, chaque machine recalcule

8. Dans notre exemple, comme la fonction $\min$ est commutative, l'ordre de réception n'a pas d'importance.

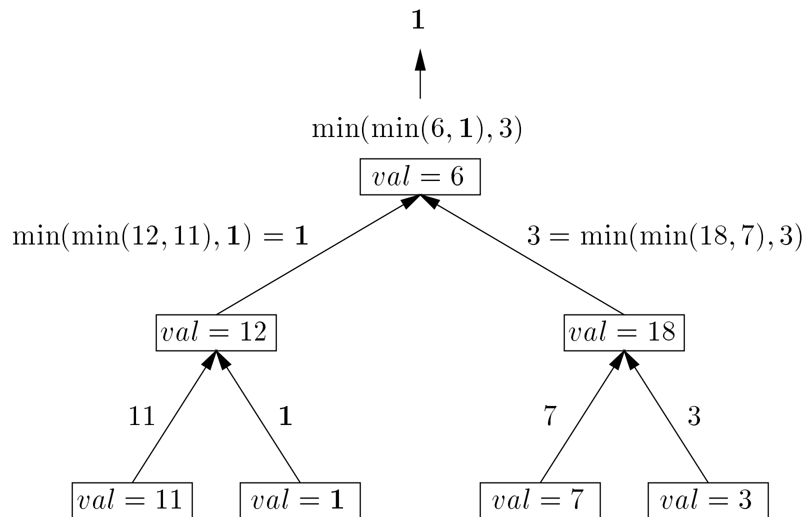


FIG. 5.6: *Exemple d'une opération de réduction calculant le minimum réparti.*

son minimum local, à partir des éventuelles valeurs lues dans le sous-groupe portant son nom, et transmet le résultat de ce calcul dans le sous-groupe portant le nom de son père.

Pour lancer l'algorithme, il suffit de lancer une instance du programme sur chacune des machines, de la racine vers les feuilles.

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#define PERIODE 1  /* 1 seconde */
#define err_exit(message) {perror(message);exit(1);}

/* Descripteur des fichiers chan */
int fd_fils,fd_pere;

/* Calcul de la charge locale a partir de la commande uptime */
float charge_locale()
{
  FILE *cmd_file;
  float charge=0.0;

  if ((cmd_file = popen("uptime | cut -d: -f4","r")) < 0)
    err_exit("obtention de la charge");

  if (!fscanf(cmd_file,"%f,",&charge))
    {fprintf(stderr,"lecture charge impossible\n");exit(1);}

  pclose(cmd_file);
  return charge;
```

```
}

/* Initialisation des groupes et sous-groupes */
void initialisation(char *parent,char *mon_nom)
{
  int fd_reg,fd_act;
  char buf[128];

  /* Enregistrement dans le groupe “un-groupe” */
  if ((fd_reg= open("/mpc/register",O_WRONLY)) < 0)
    err_exit("ouverture register");
  if (write(fd_reg,"+un_groupe",strlen("+un_groupe")) < 0)
    err_exit("enregistrement");
  close(fd_reg);

  /* Association de la machine du parent */
  if (parent){
    if ((fd_act= open("/mpc/un_groupe/activate",O_WRONLY)) < 0)
      err_exit("ouverture activate");
    if (write(fd_act,parent,strlen(parent)) < 0)
      err_exit("association du pere");
    close(fd_act);
  }

  /* Deplacement vers le repertoire propre (sous-groupe racine) avec le lien self */
  if (chdir("/mpc/un_groupe/self/") < 0)
    err_exit("sous-groupe racine");

  /* Creation du ou des sous-groupes */
  if (parent && (mkdir(parent,S_IRWXU) < 0))
    err_exit("creation sous-groupe parent");
  if (mkdir(mon_nom,S_IRWXU) < 0)
    err_exit("creation sous-groupe fils");

  /* Ouverture des fichiers chan ( ./<parent>/chan et ./<mon_nom>/chan ) */
  if (parent){
    sprintf(buf,"%s/chan",parent);
    if ((fd_pere=open(buf,O_WRONLY)) < 0)
      err_exit("ouverture vers parent");
  }
  sprintf(buf,"%s/chan",mon_nom);
  if ((fd_fils=open(buf,O_RDONLY)) < 0)
    err_exit("ouverture depuis fils");

}

main(int argc, char **argv)
{
  char *parent=NULL;
  char mon_nom[MAXHOSTNAMELEN];
  char buf[128];

  if (argc == 2) parent=argv[1];
  gethostname(mon_nom,MAXHOSTNAMELEN);
```

```
  initialisation(parent,mon_nom);

  while (1){
    char nom_min[MAXHOSTNAMELEN]; /* Le nom de la machine ayant le min */
    float min=charge_locale();    /* calul de la valeur locale */
    fd_set fds;
    struct timeval tv={0,0};
    int pret;
    int str_len;

    strcpy(nom_min,mon_nom);
    sleep(PERIODE);
    FD_ZERO(&fds);
    FD_SET(fd_fils,&fds);

    /*
     * Lecture des donnees envoyees par les fils.
     * Les nouveaux messages sont detectes avec select()
     * On pourrait aussi basculer le descripteur en mode non bloquant
     */
    while((pret=select(fd_fils+1,&fds,NULL,NULL,&tv)) >0){
      float fils_min;
      char nom_fils[MAXHOSTNAMELEN];

      if (read(fd_fils,buf,128) < 0)
        err_exit("lecture fils");

      /* Les messages sont de la forme "machine x.xx" */
      if (sscanf(buf,"%s %f",nom_fils,&fils_min) != 2)
        {fprintf(stderr,"message malforme\n");continue;}

      /* calcul du min */
      if (fils_min < min){
        min = fils_min;
        strcpy(nom_min,nom_fils);
      }
    }

    /* Envoi du minimum vers le parent (ou affichage) */
    str_len=sprintf(buf,"%s %3.2f\n",nom_min,min);
    if (parent){
      if (write(fd_pere,buf,str_len+1) < 0)
        err_exit("ecriture pere");
    }
    else printf("%s\n",buf);
  }
}
```

5.4 Discussion

MPCFS permet de répondre de façon efficace aux besoins de communication des applications réparties s'exécutant sur un réseau de travail. Il satisfait en effet la majorité des critères de qualité que nous nous sommes imposé dans le cahier des charges établi au paragraphe 5.1.2 (page 98) :

- **performant** : le système est implémenté au cœur du noyau UNIX. Ce contexte d'exécution privilégié lui permet, par exemple, d'être (potentiellement) plus performant que les solutions proposées en contexte utilisateur, qui, de leur côté, ont généralement à souffrir de multiples changements de contexte et de recopies inutiles de messages en mémoire[9]. La solution proposée permet en particulier d'optimiser le chemin critique de traitement des messages ;

- **efficace** : l'utilisation de ce système permet d'éviter de gaspiller inutilement les ressources du système et du réseau, (i) en minimisant le temps de traîtement des messages, (ii) en accédant si besoin est, à tous les niveaux des couches de protocoles, (iii) en évitant la redondance des mécanismes de gestion des communications et des groupes de communication dûs aux exécutions simultanées de plusieurs applications réparties, ou encore, (iv) en ayant la possibilité de mettre en place des stratégies de contrôle de flux centralisées efficaces ;

- **multi-utilisateurs et multi-tâches** : ce mécanisme supporte évidemment d'être utilisé simultanément par plusieurs processus et applications ;

- **équitable** : la concurrence entre les différents processus qui ont simultanément recours à ce mécanisme est gérée, comme chacune des ressources contrôlées par le noyau UNIX, de façon à assurer à chacun d'eux une qualité de service comparable ;

- **portable** : dès lors que MPCFS est disponible sur plusieurs systèmes UNIX, la portabilité des applications produites est immédiate et surtout, garantie à long terme. L'API utilisée (le système de fichiers) est en effet l'une des plus stables et uniformément généralisée dans les systèmes UNIX. Bien-sûr cette portabilité reste conditionnée par la disponibilité du mécanisme dans les systèmes UNIX existants. Celle-ci est raisonnablement envisageable sur la majorité des systèmes récents, qui adoptent quasiment tous l'architecture de *VFS* sur laquelle s'appuie MPCFS ;

- **transparent** : MPCFS a été conçu de façon à dissimuler les spécificités du système UNIX et du ou des réseaux de communication sous-jascents. Toutes les fonctionnalités proposées sont accessibles par des opérations portant sur des fichiers et répertoires, lesquelles sont toutes disponibles au travers de la même API sur chacun des UNIX existants ;

9. Certains travaux récents [Welsh 97, Koch 98] montrent qu'il est possible d'atteindre des performances optimales en contexte utilisateur, sans passer par l'intermédiaire de noyau. Les solutions proposées impliquent toutefois une réservation exclusive du medium de communication par les processus utilisateurs.

– **robuste** : le fonctionnement totalement réparti de ce mécanisme, qui ne repose sur aucun service centralisé, lui permet facilement de tolérer les pannes de machines ou de réseau ;

– **fonctionnel** : sur le plan des fonctionnalités, le système MPCFS, tel qu'il est présenté, ne supporte que des communications multipoints fiables, mais non atomiques et simplement ordonnées selon la source. Les possibilités d'évolution et d'extension de ce système sont toutefois pratiquement illimitées. On peut ainsi aisément prévoir l'ajout de nouvelles fonctionnalités de communication par la simple adjonction de fichiers virtuels au système de fichiers actuel ;

– **convivial** : l'API proposée est très simple d'utilisation, et intuitive. Elle est compatible avec l'ensemble des langages et outils existants dans les systèmes UNIX et propose des fonctionnalités permettant de faciliter le déboguage et la mise au point des applications (visibilité des messages en transit, toutes les fonctionnalités sont ouvertes à tous les processus) ;

– **évolutif** : la mise en œuvre de ce système au niveau du noyau, et l'indépendance de l'API proposée vis-à-vis des réseaux et protocoles sous-jascents, permet d'envisager un support transparent des technologies et protocoles futurs ;

– **sécurisé** : les modalités d'utilisation par les utilisateurs de même que la spécification des ensembles de machines autorisées à coopérer sont entièrement placées sous le contrôle de l'administrateur de la machine ;

– **confidentiel** : au travers des permissions sur les fichiers et répertoires de son arborescence, MPCFS permet à une application (resp. un utilisateur) d'autoriser ou d'interdire la lecture à son insu de ses messages par d'autres applications (resp. utilisateurs).

MPCFS permet donc de combiner des qualités que l'on trouve dans les systèmes existants, mais parfois de façon exclusive. Les solutions proposées dans les systèmes répartis sont, par exemple, généralement performantes, efficaces, multi-tâches et multi-utilisateurs, mais peu portables, à l'inverse des solutions proposées en contexte utilisateur par les environnements de programmation et bibliothèques de communication.

Naturellement, MPCFS comporte des défauts :

– **choix d'un type de diffusion** : le système ne supporte qu'un seul type de diffusion, dont le niveau de fiabilité et les propriétés de séquencement peuvent s'avérer insuffisantes pour bon nombre d'application. Ce défaut peut toutefois être corrigé par l'introduction de nouveaux nœuds de communication dans chaque répertoire de multiplexage, assurant chacun des niveaux de qualité de service différents : par exemple un fichier de nom `chan_atom_total` pour une diffusion atomique totalement ordonnée, un autre de nom `chan_atom_causal` pour une diffusion atomique causale, etc.

– **informations concernant les machines associées** : aucun mécanisme n'est proposé pour récuperer les informations qui concernent les machines associées à un groupe

(autre que celle, minimale, obtenue au travers du fichier `activate`). Il serait intéressant, par exemple, de pouvoir visualiser les parties de l'arborescence des sous-groupes qui sont communes à la machine locale et à chaque machine associée.

– **absence de mécanisme de routage** : en dépit des avantages que nous avons recensés au paragraphe 5.2.2.1, cette absence peut dans certaines situations s'avérer problématique. La prise en charge de ce routage par des processus, en contexte utilisateur, par exemple, n'est pas efficace en terme de latence (elle implique des changements de contexte et des recopies inutiles des messages routés). Proposer un mécanisme permettant aux utilisateurs de faire prendre en charge de façon automatique cette redirection, à partir du contexte du noyau, serait par exemple souhaitable.

– **gestion de la mémoire** : la politique de gestion de la mémoire de stockage, utilisée pour conserver les messages en transit, n'est pas assez souple.

– **choix d'un groupe de communication** : dans l'état actuel du système, l'utilisateur est obligé de consulter (et d'interpreter) les fichiers de configuration du système pour choisir un groupe qui ait les propriétés qu'il désire (propriétaire, mode de fonctionnemnet, machines autorisées, . . .). Pour simplifier cette tâche, qui peut s'avérer assez complexe, il serait souhaitable de lui proposer un moyen de séléctionner un nom de groupe de façon automatique, selon des critères de séléction génériques.

– **association de groupes de même nom** : le système ne permet actuellement que les associations entre des groupes de processus ayant des noms logiques identiques sur chacune des machines. Il peut toutefois s'avérer intéressant, dans certaines situations, de permettre l'association de groupes portant des noms différents sur chacune des machines. Rappelons en effet, que les permissions et les modes de fonctionnement de chacun des groupes sont fixés au niveau de chaque machine, en fonctions des noms logiques de groupes. Lorsque la configuration des groupes sur chacune des machines n'est pas la même, ou simplement lorsque la configuration du système est différente (par exemple, un utilisateur peut avoir des noms de *login* différents selon les machines), il peut s'avérer impossible pour un utilisateur de créer sur chacune des machines qu'il souhaite utiliser, le même groupe de communication, avec les même paramètres de fonctionnement.

– **informations concernant l'activité du groupe** : il manque un mécanisme permettant aux applications d'avoir des informations concernant l'activité du groupe en temps réel, et de façon passive : enregistrement et dés-enregistrement de processus locaux, association et séparation de machines, transmission résussies de messages, etc. A l'exception des transmissions réussies de messages, ces informations peuvent actuellement être facilement obtenues, mais de façon active (*polling*), c'est-à-dire en listant le répertoire racine du groupe ou en lisant le fichier `activate`. Une méthode passive et différentielle (ne rendant compte que des modifications subies) serait intéressante.

5.5 Conclusion

Dans ce chapitre, nous avons présenté le système de fichiers MPCFS, un mécanisme de communication multipoints original pour les systèmes UNIX.

Comme l'a montré la discussion du paragraphe 5.4, l'approche choisie présente de nombreux avantages, que l'on ne retrouve pas tous simultanément dans les autres solutions existantes. Cette discussion montre aussi que le système MPCFS peut encore être amélioré, et que de nombreuses fonctionnalités peuvent encore être ajoutées, comme la diffusion atomique. Notre objectif avec ce travail n'était toutefois pas de décrire une solution complète, proposant d'emblée tous les services que l'on peut espérer d'un tel système. La spécification complète d'un tel système et surtout, son indispensable validation expérimentale au travers d'un prototype, est en effet un projet dont l'envergure dépasse largement le cadre d'une thèse. Au contraire, notre objectif, était plutôt de démontrer, au travers d'un ensemble minimal de fonctionnalités, la faisabilité d'un tel système, et l'intérêt de l'approche proposée par rapport aux autres solutions que sont les environnements de programmation ou les systèmes d'exploitation répartis.

Pour cela, nous avons développé un prototype de ce système pour le système Linux, implémentant la presque totalité des fonctionnalités que nous avons décrites dans les paragraphes précédents. Ce prototype est largement décrit dans les derniers chapitres de cette thèse.

Précisons enfin que ce système n'a pas la prétention de résoudre tous les problèmes de communication auxquels doivent faire face les applications réparties. En particulier, il n'apporte (et ne peut apporter) aucune solution nouvelle dans le domaine de l'hétérogenéité : son utilisation entre machines d'architectures différentes se heurte à l'éternel problème de la représentation interne des données, qui peut varier d'une machine à l'autre. L'approche MPCFS a donc ses limites, qui sur certains points peuvent paraître se situer en-deça de celles des autres solutions. En effet, un environnement tel que PVM, par exemple, gère ce problème d'hétérogenéité de façon transparente. Mais dans l'absolu, ces limites ne remettent pas en cause l'efficacité du système, dont les fonctionnalités peuvent toujours être complétées par l'ajout de surcouches de plus haut niveau.

Ainsi, par exemple, réécrire un environnement tel que PVM au dessus du système MPCFS, pourrait s'avérer très avantageux : le système résultant serait potentiellement plus performant en matière de communications multipoints, il pourrait inter-opérer de façon transparente avec tout autre application ayant aussi recours aux services de MPCFS (un environnement de répartition dynamique de charge, par exemple), ou encore, il permettrait une mise au point plus facile des applications réparties, qui bénéficieraient des facilités offertes par MPCFS, telle que la visualisation des messages en transit.

Troisième partie

Mise en oeuvre d'un prototype de MPCFS pour le système Linux

Chapitre 6

Structures de données

6.1 Introduction

Nous débutons la description de la mise en œuvre du système de fichiers MPCFS avec ce chapitre, qui présente les principales structures données utilisées dans le prototype. Cette description, qui est assez détaillée, a pour objectif d'introduire les principaux éléments de l'architecture logicielle du système MPCFS. L'étude de ces structures de données est donc surtout intéressante afin de faciliter la compréhension des algorithmes et protocoles présentés dans les chapitres qui suivent.

Avant de commencer cette description, rappelons brièvement les contraintes que nous avons du respecter concernant l'utilisation de la mémoire. Ces contraintes découlent d'une part de notre choix d'un niveau de programmation très bas, dans le contexte du noyau, et d'autre part de notre soucis d'optimiser les performances du système MPCFS.

Comme la mémoire est une ressource critique dans le contexte du noyau, son utilisation doit être minimisée. D'autre part, l'allocation dynamique de la mémoire est d'autant plus coûteuse que la quantité de mémoire demandée est importante. Une bonne politique d'allocation doit donc favoriser l'allocation temporaire de petites zones de mémoire, au fur et à mesure des besoins, plutôt que l'allocation de grandes zones de mémoire sur une longue durée. Par exemple, cette stratégie nous a souvent conduit à utiliser des structures de type liste là où des structures de types tableau pourraient paraître plus efficaces.

Cependant, pour des raisons de performance, nous avons parfois préféré des structures coûteuses en termes d'allocation, par exemple afin de favoriser le chemin critique de traitement des messages.

Nous commençons par décrire la stratégie de numérotation des i-nœuds du système de fichiers. Puis nous décrivons successivement les structures de données proprement dites : la structure associée aux groupes de communication, la structure associée aux sous-groupes, les structures associées aux processus membres d'un groupe, la structure associée aux connexions, la structure des associations et, pour finir, la structure associée aux messages.

6.2 Gestion des i-nœuds

Comme dans tout système de fichiers UNIX, les fichiers de l'arborescence MPCFS sont associés à un i-nœud, lequel est identifié de façon non ambiguë grâce à un numéro. Ces numéros d'i-nœuds sont des entiers sur 32 bits. La gestion de ces numéros d'i-nœuds est prise en charge par le pilote de périphérique à partir du point de montage et pour toute l'arborescence dont il a la charge. Le pilote du système de fichiers MPCFS gère donc lui-même son propre espace d'adressage (sur 32 bits) pour les i-nœuds du système de fichiers MPCFS.

En d'autres termes, c'est au système de fichiers MPCFS qu'incombe la charge d'attribuer un i-nœud unique à chacun des répertoires et fichiers de l'arborescence MPCFS. La solution que nous avons retenue pour cette numérotation est présentée figure 6.1. Elle consiste à découper les 32 bits de l'i-nœud en 4 champs de bits : le premier désigne le type du fichier, le

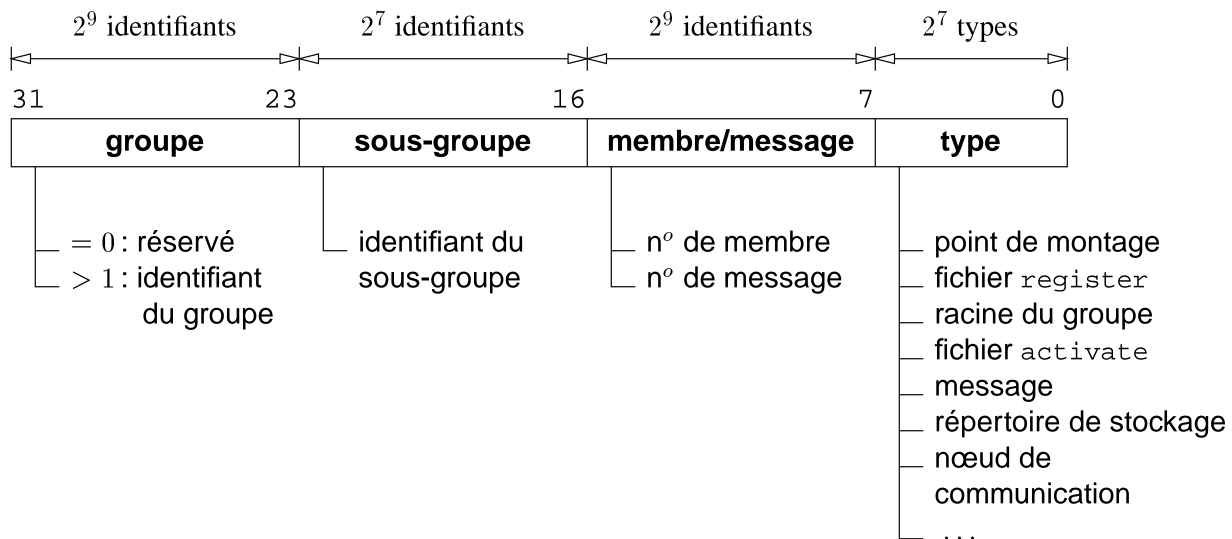


FIG. 6.1: *Numérotation des i-nœuds dans le système de fichiers MPCFS.*

second désigne un membre ou un numéro de message (selon le type de fichier), le troisième désigne le sous-groupe et enfin, le dernier désigne le groupe de communication [1].

Cette présentation faite, revenons maintenant plus en détail sur le rôle de ces i-nœuds. Comme nous l'avons dit, les i-nœuds identifient de façon unique les fichiers et répertoires d'un système de fichiers. Le VFS, l'interface générique de gestion des fichiers, manipule tous les fichiers de l'arborescence globale de la machine de la même façon, par l'intermédiaire de leur structure d'i-nœud. Pour obtenir cette structure, le VFS invoque une fonction spécifique, que doivent fournir chacun des pilotes de périphérique ayant la charge d'une partie de l'arborescence : la fonction `read_inode()` [Beck 98].

Par exemple, pour obtenir la structure d'i-nœud du répertoire racine d'un groupe de l'arborescence MPCFS, le VFS invoque la fonction `read_inode()`, du système de fichiers MPCFS, avec une structure d'i-nœud vide, ne contenant que le numéro de l'i-nœud recherché [2].

C'est alors à la fonction `read_inode()` du pilote de l'arborescence MPCFS, de déterminer, en fonction de ce seul numéro d'i-nœud, de quel fichier ou répertoire il s'agit, et de remplir la structure d'i-nœud en conséquence. Dans notre cas, cette fonction extrait les 7 bits de poids faible du numéro d'i-nœud, afin de déterminer le type du fichier. Elle détermine qu'il s'agit du répertoire racine d'un groupe de communication. Elle extrait alors les 9 bits de poids fort, et recherche la structure de groupe de communication correspondante. Ayant trouvé cette structure, elle peut remplir la structure de l'i-nœud.

1. Comme l'indique la figure, le numéro de groupe 0 est réservé afin de désigner les quelques fichiers et répertoires de l'arborescence qui n'appartiennent à aucun groupe (fichier `register`, racine du système de fichier, ...).

2. Pour obtenir ce numéro, le VFS a du, au préalable, obtenir la structure d'i-nœud du répertoire parent de ce répertoire, (en l'occurrence la racine du système de fichiers MPCFS), et réclamer son contenu auprès du pilote MPCFS.

Les i-nœuds étant codés sur 32 bits, pourquoi alors ne pas avoir simplement choisi l'adresse physique (elle aussi sur 32 bits) de chaque structure associée à un élément de l'arborescence MPCFS comme numéro d'i-nœud ? Cette méthode d'adressage direct est certes plus rapide. Mais en apparence seulement, car elle comporte plusieurs inconvénients.

Tout d'abord, la programmation au niveau du noyau exige la plus grande prudence. Si, pour une raison ou pour une autre, le noyau réclame un numéro d'i-nœud qui n'est pas attribué, cette solution conduit à retourner une adresse invalide, avec les conséquence que l'on imagine. Bien sûr, on peut toujours envisager de mettre en place un contrôle sur les adresses manipulées. Mais dans ce cas, la numérotation structurée que nous proposons n'est pas moins efficace, car la vérification est à la fois simple et fiable.

Ensuite, cette solution est dangereuse en termes de portabilité, car les adressages sur 32 bits deviennent de plus en plus rares dans les architectures de processeurs. Les systèmes de fichiers, en revanche, risquent de se satisfaire de numéros d'i-nœuds sur 32 bits pour de longues années encore.

Enfin, la numérotation structurée présente aussi des avantages. Elle permet notamment à une commande extérieure au système de fichiers MPCFS, d'interpréter les numéros d'i-nœuds et de retrouver les identifiants associés à chacune des structures, ce qui est intéressant pour aider à la mise au point du système lui même. En effet, rappelons que les numéros d'i-nœuds des fichiers ne sont pas gardés secrets par le système, mais librement consultables, à l'aide de la commande `ls -i`, par exemple.

6.3 Groupes de communication

La structure représentant le groupe de communication est l'élément central de l'architecture du système MPCFS, en raison du rôle que jouent ces groupes de communication dans le fonctionnement du système. Ceux-ci interviennent dans toutes les opérations, que ce soit au niveau des interactions des utilisateurs avec le système de fichiers, ou au niveau de la gestion des communications.

6.3.1 Description de la structure

Les informations contenues dans cette structure peuvent être groupées en différentes catégories, reflétant les multiples rôles des groupes de communication au sein du système. Nous distinguons les sept catégories suivantes :

1. **Identification du groupe de communication**. On distingue les principaux champs suivants :

   – l'**identifiant de groupe** : chaque groupe de communication existant possède un identifiant local unique. Il permet de désigner et de retrouver très rapidement la structure de données qui lui est associée (voir section 6.3.2) ;

   – l'**identifiant de génération** : comme les identifiants sont réattribués au cours du temps, une machine extérieure peut confondre un groupe d'identifiant A avec le

  groupe qui était associé à A dans le passé. Cet identifiant de génération, qui est renouvelé à chaque attribution d'un identifiant de groupe, permet donc de lever cette ambiguïté ;
- le **nom du groupe** : cette chaîne de caractères représente le nom apparent du groupe, c'est-à-dire le nom du répertoire hébergeant la racine de l'arborescence de ce groupe de communication dans le système de fichiers.

2. **Composition du groupe** : il s'agit de pointeurs vers les trois listes d'éléments qui constituent un groupe de communication :
   - la **liste des membres** contient les structures associées à chacun des processus qui appartient au groupe de communication.
   - la **liste des associations** contient les structures qui décrivent les associations du groupe avec ceux des autres machines.
   - la **liste des sous-groupes** contient les structures qui décrivent les sous-groupes du groupe de communication. Le sous-groupe racine apparaît toujours en tête de cette liste.
3. **État du groupe** : cette catégorie regroupe les informations reflétant l'état du groupe de communication, telles que les drapeaux d'états, les nombres de membres, d'associations, de sous-groupes, de messages en attente de lecture, ainsi que la quantité de mémoire utilisée pour le groupe.
4. **Limites du groupe** : ce sont les limites imposées au groupe de communication lors de sa création. Elles concernent l'occupation mémoire (globale et par répertoire de stockage), le nombre de messages (global et par répertoire de stockage), et la taille de l'arborescence des sous-groupes.
5. **Historique du groupe** : il s'agit par exemple des nombres de messages échangés, envoyés et reçus et des nombres d'octets correspondants.
6. **Permissions et attributs** : ce sont d'une part, les permissions et attributs affectés par défaut lors de l'apparition d'un nouvel élément dans l'arborescence du groupe et d'autre part, les autorisations de modifier ces permissions (voir paragraphe 5.3.2.9, page 123 et annexe B.1, page 238).

6.3.2 Accès aux groupes de communication

La structure de donnée qui décrit les groupes de communication est très souvent sollicitée, car elle constitue le point de passage obligatoire lors de la recherche des autres structures de données dépendant d'un groupe de communication. L'accès à cette structure doit donc être aussi rapide que possible, mais sans pour autant conduire à un gaspillage inutile de la mémoire.

Nous avons écarté l'accès direct ou même l'accès par simple indirection car ces solutions nous obligent à pré-allouer de façon statique la mémoire nécessaire afin de supporter le

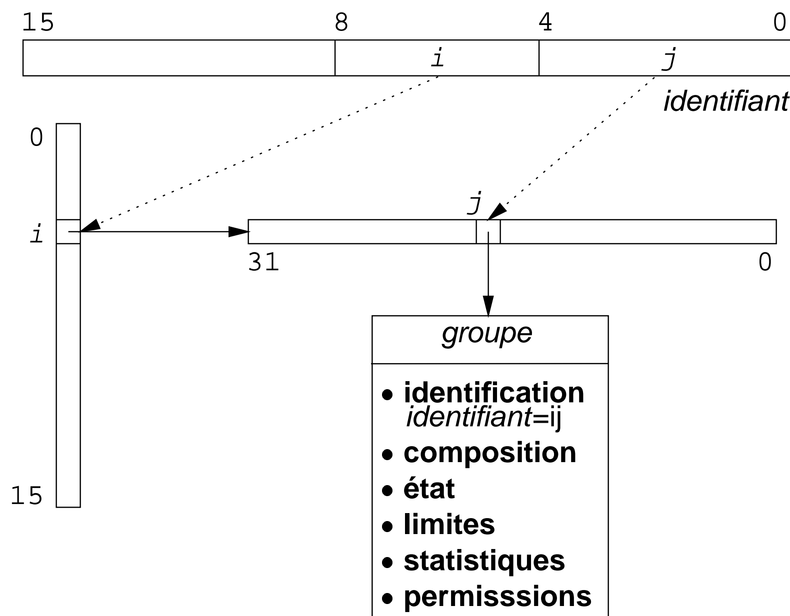


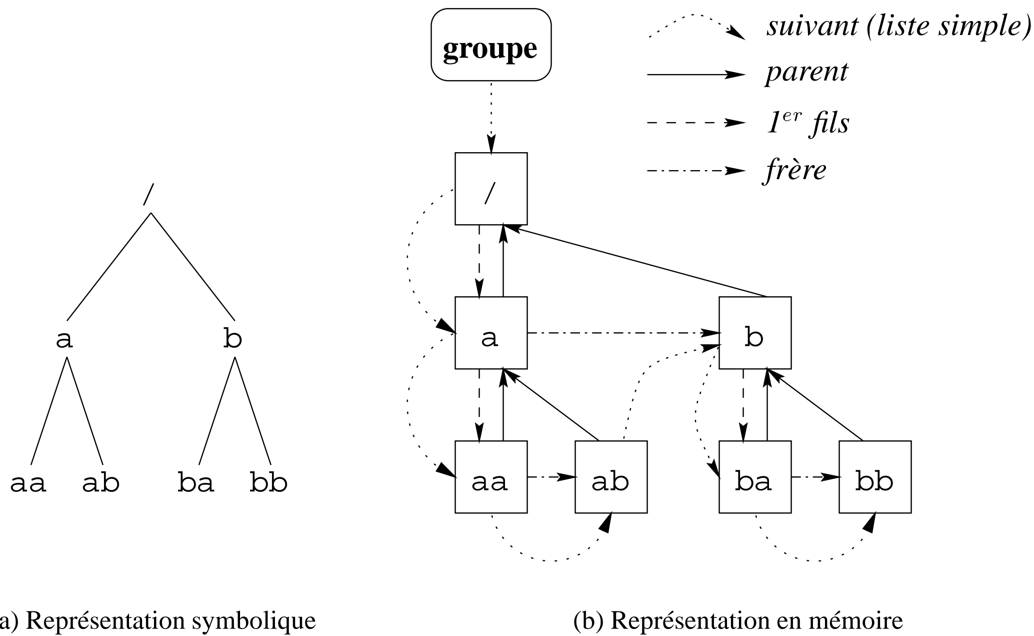
FIG. 6.2: *Accès à la structure d'un groupe de communication*

nombre maximal de groupes de communication (de 2 à 20 ko pour un nombre maximal de groupes fixé à 512), alors que seule une petite partie de cette mémoire est utilisée en général.

Partant du constat qu'un système qui héberge plus de trois ou quatre applications actives devient rapidement chargé, et d'autre part, du principe qu'une application répartie n'aurait que rarement l'usage de plus d'un groupe de communication à la fois, il nous a paru judicieux de considérer qu'en fonctionnement normal, le système n'aurait que rarement l'occasion d'héberger plus d'une dizaine de groupes de communications. Il faut néanmoins permettre au système de supporter, à l'occasion, un nombre important de groupes de communication.

En conséquence, nous avons opté pour un adressage à double indirection, dans lequel les éléments du deuxième niveau permettent d'adresser 32 groupes de communication, pour une taille effective de seulement 128 octets. Avec un maximum de groupes fixé à 512, l'élément de premier niveau ne comporte lui que 16 entrées. L'élément du premier niveau et le premier des éléments de second niveau sont alloués de façon statique lors de l'initialisation du système. Les autres éléments du second niveau sont ensuite alloués dynamiquement en cas de besoin (lorsque le nombre des groupes actifs dépasse un multiple de 32).

Comme nous l'avons vu au paragraphe précédent, les groupes de communication possèdent chacun un **identifiant de groupe** unique, formé d'un entier sur 16 bits. Avec un nombre maximal de groupe fixé à 512, seuls les neufs bits de poids faible sont significatifs. Comme le montre la figure 6.2, les cinq bits de poids faible de cet identifiant donnent la position de l'adresse de la structure dans l'élément de second niveau. Les quatre bits suivants donnent la position de l'adresse de cet élément de second niveau dans l'élément du premier niveau.



(a) Représentation symbolique (b) Représentation en mémoire

FIG. 6.3: *Liens entre sous-groupes*

6.4 Sous-groupes

La structure utilisée pour décrire les sous-groupes de communication est assez complexe. Nous distinguons huit catégories de champs dans cette structure :

1. **Identification du sous-groupe** : de la même façon qu'avec les groupes de communication, les principaux champs que l'on trouve dans cette catégorie sont les suivants :
   - l'**identifiant de sous-groupe dans le groupe** (16 bits) ;
   - l'**identifiant de génération** (idem groupes, cf paragraphe 6.3)
   - le **nom du sous-groupe** (idem groupes).
2. les **liens du sous-groupe avec les autres sous-groupes** : ce sont les champs utilisés pour construire et parcourir l'arborescence des sous-groupes d'un groupe de communication. Ces champs sont au nombre de cinq (figure 6.3) :
   - un pointeur vers le **sous-groupe suivant dans la liste** : cette liste, triée selon les identifiants de sous-groupes, relie entre eux chacun des sous-groupes, indépendamment de leur position dans l'arborescence. Elle est utilisée afin d'accélérer le parcours itératif de tous les sous-groupes d'un groupe de communication (en particulier lors de la recherche d'un sous-groupe à partir de son identifiant, afin d'en vérifier l'existence, par exemple).
   - un pointeur vers le **sous-groupe parent dans l'arborescence** ;
   - un pointeur vers le **sous-groupe frère suivant dans l'arborescence** ;

  - un pointeur vers le **premier sous-groupe fils dans l'arborescence**.
  - **le nombre des sous-groupes fils**.

3. **Gestion des messages** : les champs que l'on trouve dans cette catégorie sont utilisés lors du stockage ou de la suppression d'un message dans le répertoire de stockage associé au sous-groupe de communication. Les principaux champs de cette catégorie sont au nombre de quatre :

   - le **dernier identifiant de message attribué** : cette information permet d'attribuer les identifiants de message de façon cyclique. Cette propriété permet en général à l'utilisateur de consulter les messages dans le répertoire de stockage selon leur ordre d'apparition.
   - un pointeur vers une **structure de message sentinelle** : l'utilisation d'une structure sentinelle (c'est-à-dire pré-allouée) facilite le traitement pour les processus endormis sur une lecture bloquante lors de la réception d'un nouveau message : lorsqu'un processus endormi se réveille, la référence au message sentinelle qu'il avait avant de s'endormir est devenue la référence sur son premier message en attente de lecture (pour plus de détails, se reporter à la description des algorithmes d'envoi et de réception des messages, aux paragraphes 7.3.3, page 171 et 7.3.4, page 174).
   - un champs de bits établissant la **carte des identifiants de messages attribués**. En effet, les messages ne disparaissent pas obligatoirement du répertoire de stockage dans l'ordre où ils y apparaissent. Ceci découle des deux propriétés suivantes : (i) un processus ne reçoit pas les messages qu'il a lui-même expédiés (donc tous les processus membres d'un groupe ne consomment pas obligatoirement les mêmes messages) et (ii) les processus consomment leurs messages de façon asynchrone.
   - un entier indiquant **le nombre des messages présents dans le répertoire de stockage**.

4. **Gestion des files d'attente des processus** : chaque sous-groupe dispose de deux files d'attente, l'une pour les processus endormis dans l'attente d'un message (en cas de lecture bloquante), et l'autre pour les processus endormis dans l'attente d'une place disponible pour stocker un nouveau message (en cas d'écriture bloquante, lorsque la capacité de stockage du sous-groupe est atteinte).

5. **Compteurs de références** : chaque sous-groupe gère deux compteurs de référence, l'un comptant les référence locales (les références venant des membres du groupe sur la machine locale) et l'autre comptant les références distantes (c'est-à-dire venant de machines distantes).

6. **Informations d'état** : on trouve dans cette catégorie des informations telles que la date de dernière modification du sous-groupe (la création ou destruction de sous-groupes fils, par exemple), ou les drapeaux d'état du sous-groupe.

7. **Gestion des associations** : cette catégorie regroupe les informations concernant les interactions du sous-groupe avec les sous-groupes équivalents sur les autres machines, lorsque l'espace de stockage est épuisé (voir paragraphe 8.3.2) :

   – le **nombre de messages rejetés** : indique le nombre de messages qui n'ont pu être acceptés par l'un des sous-groupes distants par manque d'espace de stockage.

   – le **nombre d'associations bloquées** indique le nombre de machines auxquelles on a demandé d'interrompre leurs transmissions car le sous-groupe local ne dispose plus d'un espace de stockage suffisant.

   – le **nombre d'associations bloquantes** indique le nombre des machines qui ont demandé à ce que les émissions de message en provenance de la machine locale soient interrompues. Notons que l'information concernant l'identité de chacune de ces machines n'est pas conservée dans cette structure (elle peut-être obtenue en parcourant la liste des associations du groupe).

   – le numéro d'ordre de la **dernière association bénéficiaire d'un espace de stockage réservé**. Afin d'éviter les inter-bloquages lorsque les sous-groupes de plusieurs machines sont simultanément à cours d'espace disponible, le sous-groupe local ainsi que chacun des sous-groupes distants auquel il est associé, disposent d'un certain nombre d'emplacements réservés dans l'espace de stockage. Ces emplacements ne peuvent être utilisés que pour stocker des messages en provenance de la machine à laquelle ils sont attribués.

8. **Permissions et attributs** (cf. groupes).

6.5 Membres d'un groupe de communication

La structure représentant les membres d'un groupe de communication est simple. Elle sert d'une part à conserver les informations spécifiques à un membre du groupe de communication (numéro de processus, numéro de membre, permissions par défaut demandées lors de l'enregistrement). D'autre part elle est utilisée pour conserver la liste des sous-groupes du groupe de communication que le membre a créé dans son arborescence propre, et les permissions sur les répertoires associés. Les structures associées à chacun des membres d'un groupe sont reliées par une liste simple.

6.6 Connexions

La structure de données associée aux connexions est la plus complexe, car la gestion de ces connexions doit faire face à de nombreuses difficultés techniques et algorithmiques : les connexions doivent gérer le multiplexage des nombreux flots de données intra et inter-groupes qui peuvent simultanément exister entre deux machines. À cela, s'ajoute le flot des messages de contrôle propre à chaque connexion. La transmission des messages doit être assurée de façon fiable et ordonnée selon la source (FIFO), tout en restant aussi performante que possible. Les contraintes liées à la gestion d'un espace mémoire limité se traduisent par

une gestion délicate des tampons d'émission et de réception des messages dont la taille est limitée. Enfin, la gestion des connexions doit résoudre des problèmes de synchronisation et de conflits entre les divers processus qui interagissent avec les connexions (établissement des connexions, réalisation des communications, délivrance des messages utiles aux processus locaux, . . .).

6.6.1 Description de la structure

Nous décomposons cette structure selon les neufs catégories suivantes :

1. **Identification et paramètres de création** :

   – l'**identifiant local de la connexion** est un index dans la table des connexions ;

   – la **clef d'identification** (32 bits) associée à l'identifiant précédent. Cette clef, générée pseudo-aléatoirement lors de l'établissement de la connexion, confère au système un minimum de robustesse en matière de sécurité et de tolérance aux pannes ;

   – la structure **identifiant la machine distante** : cette structure contient l'identifiant de la connexion distante, la clef d'identification associée, le numéro de port de la *socket* distante, l'adresse IP de la machine distante, et son nom logique ;

   – les **permissions** concernant la connexion : elles déterminent si la transmission des messages utiles peut avoir lieu de la machine locale vers la machine distante seulement, dans l'autre sens seulement, ou bien dans les deux sens ;

   – la **date de création** de la connexion.

2. ***Socket* de réception**. Il s'agit du pointeur vers la structure du *socket* de réception.

3. **Synchronisation**. Cette catégorie regroupe les éléments qui permettent gérer la concurrence et la synchronisation entre processus :

   – la **file d'attente de l'établissement de la connexion** est utilisée par les processus lorsque la connexion est en cours d'établissement ;

   – une variable de type **"tester et positionner" (*test-and-set*)** qui est utilisée comme verrou lors de la mise à jour des files ;

   – un **compteur des références** à la structure émanant de processus endormis. Ce compteur permet d'empêcher la destruction de la structure tant qu'elle est référencée par des processus endormis ;

   – les informations concernant **les files d'attente des processus bloqués** (en écriture) quand les files de messages de la connexion sont pleines. La gestion de ces files est assez complexe. Nous avons en effet cherché à éviter de réveiller systématiquement (et inutilement) un nombre potentiellement grand de processus. Nous présentons et discutons cette politique de gestion des files d'attente au paragraphe suivant (6.6.2).

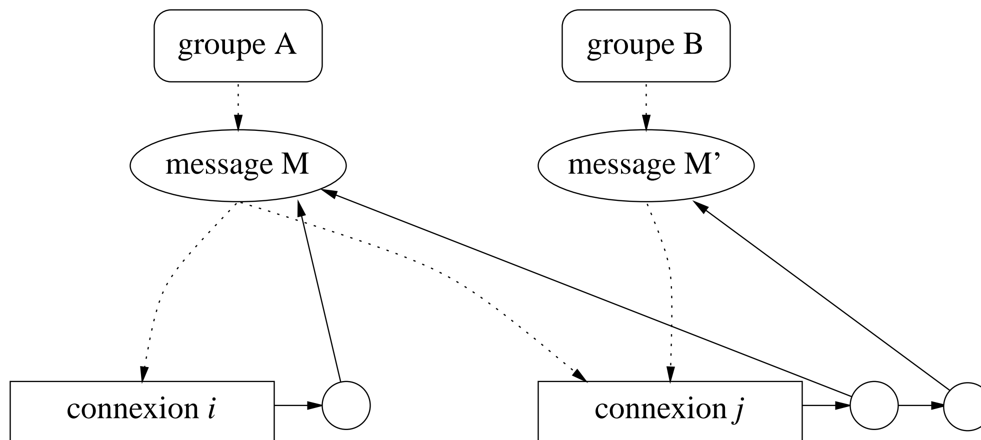


FIG. 6.4: *Accès indirect aux messages depuis les connexions*

4. **Associations**. Ce champ contient un pointeur vers la la liste des associations se partageant la connexion et le nombre d'éléments de cette liste. Pour plus de détails concernant les relations entre les structures des groupes de communication, des associations et des connexions, se reporter à la description de la structure des associations, au paragraphe 6.7 (page 155).

5. **Files de messages**. Chaque connexion gère les files de messages suivantes :

   – la file des **messages à émettre ou à ré-émettre** dès que possible ; les messages sont ordonnés (et donc émis) dans l'ordre croissant de leurs numéros de séquence ;

   – la file des **messages en cours de transmission** : cette file gère les messages dont l'acquittement n'a pas encore été reçu ; remarquons que les messages de file précédente (messages à émettre ou ré-émettre) font obligatoirement partie de cette file, mais que la réciproque n'est pas vraie ;

   – la file de **remise en ordre des messages reçus** : un messages reçu reste en attente dans cette file tant qu'un message ayant un numéro de séquence inférieur n'a pas encore été reçu. Ce mécanisme assure un séquencement de type "premier envoyé premier reçu" au niveau de chaque connexion (FIFO selon la source) ;

   – la file des **messages reçus prêts à être traités ou délivrés** : les messages de la file de remise en ordre sont déplacés sur cette file au fur et à mesure de l'arrivée des messages manquants. Notons que ces messages apparaissent sur cette file dans l'ordre où ils ont été émis par la machine expéditrice.

   Afin d'autoriser la transmission simultanée d'un même message sur plusieurs connexions (figure 6.4), les files précédentes ne relient pas les messages directement entre eux, mais font appel à de petites structures intermédiaires, référençant d'une part les messages, mais contenant aussi quelques informations à propos de la transmission

du message sur la connexion :

– le **numéro de séquence du message** sur cette connexion ;
– la **date de dernière transmissions** du message sur la connexion ;
– le **nombre de retransmission** dont le message a fait l'objet sur cette connexion ;
– un **drapeau d'état** utilisé lors de la transmission asynchrone du message par un processus tiers.

6. **Gestion des numéros de séquence** : chaque nouveau message envoyé par l'une des deux machines participant à la connexion est étiqueté à l'aide d'un numéro de séquence spécifique. Ce numéro permet de détecter les pertes de messages (les numéros de séquence sont des entiers sur 16 bits attribués cycliquement). Chacune des deux machines gère de façon indépendante les numéros de séquence des messages qu'elle envoie, ce qui autorise un fonctionnement bi-directionnel totalement asynchrone. En conséquence, la structure associée à la connexion contient les deux champs suivants :

   – le **dernier numéro de séquence envoyé** : il est incrémenté à chaque nouvel envoi de message ;
   – le **dernier numéro de séquence valide reçu** : un numéro de séquence reçu est jugé valide dès lors que tous les messages ayant les numéros de séquence qui le précédent ont été reçus.

7. **Horloges/Échéanciers** : le fonctionnement de chaque connexion est soumis à un certain nombre d'échéances, telles que la retransmission après un délai d'expiration d'un message dont l'accusé de réception n'a pas été reçu, ou l'envoi retardé de messages de contrôle. Pour cela, la structure associée à la connexion contient les champs suivants, dont les valeurs sont des dates exprimées en fonction de l'horloge système (dont le grain est de 10 ms) :

   – la **date de dernière émission** ;
   – la **date limite d'envoi d'une réponse** ; comme nous le verrons plus en détail au chapitre 8, le protocole utilisé par la couche transport dans MPCFS prévoit de retarder légèrement l'envoi d'un accusé de réception quand aucun message utile n'est prêt à être émis dans l'autre direction. Ceci permet d'éviter de générer inutilement des messages de contrôle dans le cas d'un flot de messages bi-directionnel. Ce retard doit néanmoins être contrôlé très précisément afin de ne pas provoquer l'expiration du délai de garde de retransmission sur la machine distante.
   – la **prochaine date de ré-examen de la connexion** ; cette date fixe le délai maximum au bout duquel la connexion doit être réexaminée. Ce mécanisme est utilisé en particulier afin de traiter l'expiration des délais de retransmission des messages.

8. **Gestion des retransmissions et contrôle de flux** : de façon assez similaire au protocole point-à-point TCP, le protocole mis en œuvre au niveau de la couche de transport

de MPCFS utilise d'une part, une estimation en temps réel du temps d'aller-retour, et d'autre part un mécanisme de fenêtrage. Ceci permet de déterminer le délai de retransmission des messages. Les informations nécessaires pour cela sont les suivantes :

- le **temps d'aller-retour moyen** estimé à partir de la différence entre les dates d'émission d'un message et de réception de son accusé de réception (en l'absence de pertes ou de retransmissions) ;
- la **variance du temps d'aller-retour** ;
- le **délai de garde courant** au bout duquel une retransmission doit être déclenchée ;
- le **nombre de transmissions successives sans échec** : lorsque ce nombre atteint la taille de la fenêtre courante (nombre de messages en transit), il est remis à zéro et la taille de la fenêtre est augmentée (de façon incrémentale ou géométrique, selon un algorithme de type *slow-start* [Jacobson 88]).

9. **Informations diverses concernant l'état de la connexion**. Cette catégorie regroupe les champs suivants :

- l'**état de l'établissement de la connexion** (en cours d'établissement, établie, en cours de ré-initialisation, etc) ;
- un **drapeau général d'état**, utilisé lorsque la connexion est établie pour indiquer si la connexion est bloquée (en lecture, écriture, ou par manque de mémoire), si une retransmission ou l'envoi un accusé de réception sont nécessaires, etc ;
- un **compteur de messages de continuation** qui n'ont pu être envoyés. Le rôle de ces messages est de débloquer une machine distante lorsque de l'espace devient disponible pour accueillir de nouveaux messages dans un sous-groupe de communication. Ils sont jugés prioritaires. Tant que de tels messages restent en suspend, tout nouvel emplacement disponible dans la file d'émission des messages de la connexion leur est attribué en priorité ;
- des **compteurs de blocages successifs** : ils s'agit de blocages ayant empêché la transmission ou le traitement d'un message. Ces blocages peuvent être dûs, soit à un défaut d'allocation dynamique de mémoire, soit à un manque d'espace sur les files de messages ou à une erreur sur le *socket* d'émission/réception. Lorsque le nombre de blocages dépasse une certaine limite, la connexion est réinitialisée ;
- la **date de dernier examen de la connexion**.

6.6.2 Gestion des files de processus bloqués en écriture

Le temps de transmission d'un message sur le réseau est a priori beaucoup plus long que le temps requis pour que deux processus distincts réalisent une opération d'écriture locale sur un nœud de communication. En conséquence, lorsque sur une machine, plusieurs processus envoient régulièrement des messages dans un même sous-groupe, ce sous-groupe a tendance

à se remplir rapidement. Or, la capacité de stockage des messages au niveau de chaque sous-groupe est limitée. Lorsque la limite est atteinte et que les processus ont choisi un mode transmission bloquant en écriture, ils doivent être endormis.

Le problème qui se pose alors est celui du réveil de ces processus lorsqu'un emplacement de message se libère dans le sous-groupe. En effet, la dynamique du sous-groupe fait que les emplacements sont libérés un à un, au fur et à mesure que les messages sont transmis. Il n'est donc pas souhaitable de réveiller tous les processus endormis, puisque seul l'un d'entre eux peut être satisfait. Plus encore, réveiller tous les processus se traduirait par une surcharge inutile du système lorsque les processus endormis sont nombreux. Dans ce cas, chacun d'entre eux doit effectivement vérifier s'il est le bénéficiaire de l'emplacement libéré ou se rendormir dans le cas contraire, ce qui aboutit à de nombreux changements de contexte inutiles.

Une solution possible pour résoudre ce problème consiste à définir plusieurs files d'attente et à les ordonner de façon cyclique. Lorsque plusieurs processus envoient successivement des messages, ils sont mis en attente sur des files consécutives. Lorsqu'un emplacement est libéré, les processus de la file suivante dans le cycle sont réveillés. De cette façon, si N est le nombre de files et n le nombre de processus, au plus $\lceil \frac{n}{N} \rceil$ processus sont réveillés. Clairement, plus N est grand, plus le nombre de processus inutilement réveillés est minimisé.

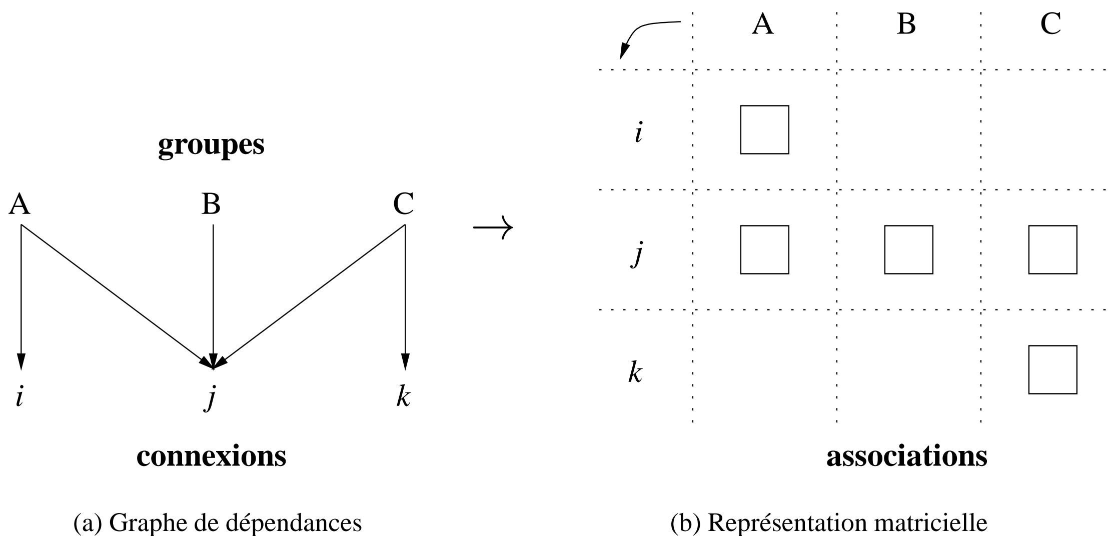
D'un autre coté, la mémoire est une ressource critique dont il faut minimiser l'utilisation. Définir systématiquement un grand nombre de files d'attente dans chaque groupe ou, a fortiori dans chaque sous-groupe, n'est donc pas raisonnable. On peut alors envisager de définir un nombre variable mais borné de files d'attente. Cette solution complique toutefois la gestion des files d'attente, en particulier lorsque le le nombre des processus bloqués est inférieur au nombre maximal de files.

La solution que nous avons retenue consiste à partager l'ensemble de ces files d'attente au niveau global, entre tous les groupes de communications. En effet, le nombre des processus qui sont en concurrence pour l'envoi d'un message est nécessairement inférieur au nombre des processus actifs à un instant donné dans le système. De plus, le nombre de processus actifs est limité par la capacité de traitement de la machine. En conséquence, quand un emplacement de message est libéré, il est possible que les processus de plusieurs groupes et sous-groupes soit réveillés.

Au pire, si tous les processus en attente se trouvent sur la même file, ils sont réveillés. Mais ce scénario a d'autant moins de chance de se produire que le nombre de files est important. En effet, pour que tous les processus soient sur la même file, il faut (i) qu'ils aient envoyé leur message au travers de groupes différents et (ii) que chacun des sous-groupes soit en phase dans le cycle d'attribution des files. Statistiquement, ce dernier point n'a que N^{-n} chances de se produire[3].

Dans le cas général, la distribution des processus en attente sur l'ensemble des files peut raisonnablement être considérée uniforme, ce qui signifie qu'en moyenne, seuls $\lceil \frac{n}{N} \rceil$ processus sont réveillés.

3. En supposant que la distribution des choix de la première file utilisées par chaque sous-groupe est uniforme sur l'ensemble des files.



(a) Graphe de dépendances (b) Représentation matricielle

FIG. 6.5: *Associations : des structures intermédiaires entre groupes et connexions*

6.7 Associations

La structure d'association n'est pas très complexe, mais c'est un élément clef dans l'architecture du système. C'est pourquoi nous détaillons son rôle au paragraphe 6.7.1 avant de la décrire au paragraphe 6.7.2, page 157.

6.7.1 Rôle des associations

Les associations sont des structures intermédiaires, qui font le lien entre les groupes de communication et les connexions au travers desquelles ces derniers envoient ou reçoivent des messages. Chacune des relations existant dans le système, entre un groupe de communication et une connexion, dispose de sa propre structure d'association. Ainsi, chaque association, qui ne référence qu'un seul groupe de communication et qu'un seule connexion, est l'emplacement privilégié pour placer les informations spécifiques à chacun des couples {groupe de communication, connexion}. En effet, d'un coté les groupes de communication peuvent être associés à plusieurs connexions (autant que de machines destinataires de leurs messages) et de l'autre, les connexions peuvent être partagées par plusieurs groupes de communication (figure 6.5).

Les associations sont des structures très importantes sur le plan des performances. En effet, elles sont placées sur le chemin critique de traitement des messages à la fois dans le sens ascendant (lorsqu'un message reçu sur une connexion doit être délivré à ses destinataires dans un groupe de communication) et, à la fois dans le sens descendant (lorsqu'un message transmis par un processus local dans un groupe de communication doit être envoyé sur chacune des connexions).

Dans le sens ascendant, l'association est retrouvée à partir de son identifiant, placé dans l'en-tête du message (figure 6.6). Dans le sens descendant, les associations sont récupérées au niveau du groupe de communication expéditeur, dans une liste chaînée (figure 6.7).

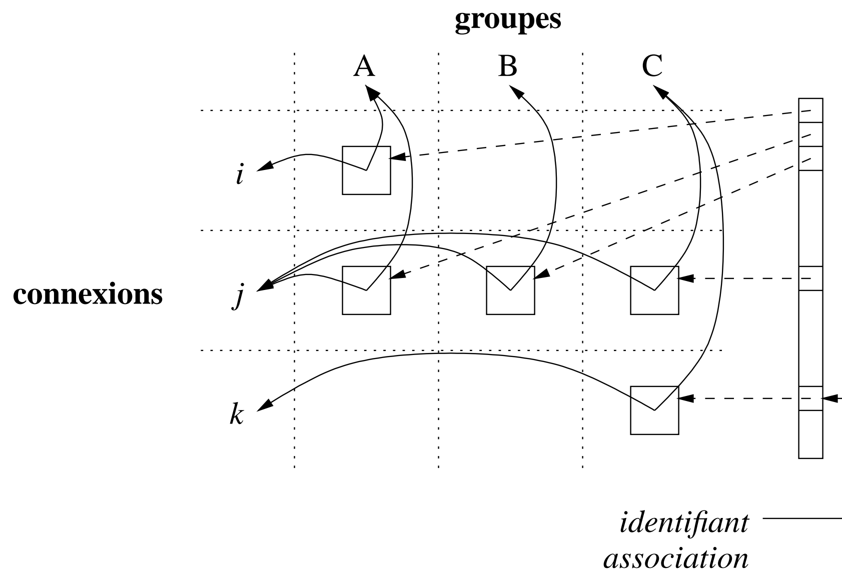


FIG. 6.6: *Accès aux associations à partir de leur identifiant*

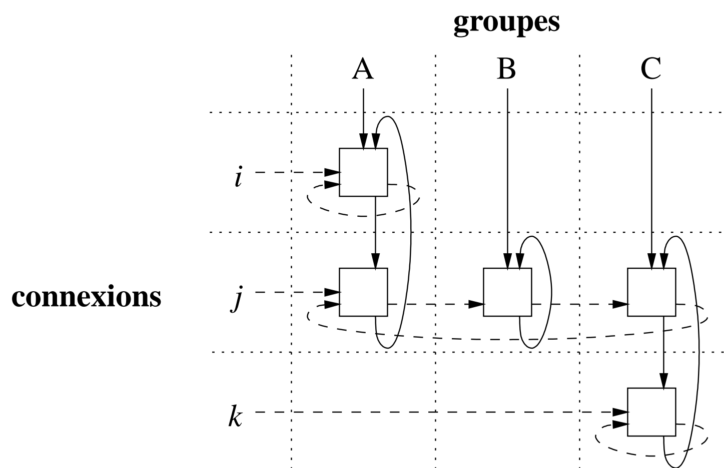


FIG. 6.7: *Accès aux associations à partir d'un groupe ou d'une connexion*

Les associations ont une autre fonction très importante, celle d'assurer la conversion entre les identifiants des sous-groupes locaux de leur groupe de communication et les identifiants de ces sous-groupes dans le groupe de la machine distante. En effet, afin respecter une stratégie de fonctionnement purement répartie, chaque machine numérote ses sous-groupes de façon totalement indépendante. La solution retenue pour permettre à chacune des machines de résoudre ce problème d'identification est d'adopter une stratégie dite "à conservation d'état" : chacune des machines est constamment tenue informée des numérotations utilisées par chacune des autres machines avec lesquelles elle est en relation, au travers d'un protocole de calcul réparti d'intersections des arborescences de sous-groupes (voir description de l'algorithme section 8.3.5, page 212). Ces informations sont conservées dans des tables de conversion, qui sont consultées lors de la réception d'un message. Étant donné que dans chaque groupe de communication, une telle table est nécessaire pour chacune des connexions utilisée, ces tables de conversion sont donc très logiquement conservées au niveau de la structure d'association.

6.7.2 Contenu de la structure

La structure d'association est formée des champs suivants :

1. des **identifiants** : l'identifiant local et son symétrique sur la machine distante, formés chacun :
   - d'un entier correspondant à la position de la référence à la structure de l'association dans un vecteur d'adresses (figure 6.6) ;
   - d'un identifiant de génération (entier, 32 bits), généré pseudo-aléatoirement lors de la création de l'association.
2. la **date de création** ;
3. des **liens** (ou références) :
   - vers la structure du **Groupe de communication** ;
   - vers la structure de la **Connexion** ;
   - vers l'association suivante dans la **liste chaînée des groupes de communication** de sa connexion (liens horizontaux sur la figure 6.7) ;
   - vers l'association suivante dans la **liste chaînée des connexions** de son groupe de communication (liens verticaux sur la figure 6.7) ;
   - vers **sa table de conversion**.
4. des **informations d'état** :
   - un **drapeau d'état** indiquant si l'association est prête à être utilisée pour acheminer des messages ;
   - le **nombre de messages de continuation en attente** ;
   - le **nombre de messages rejetés** faute de place dans les répertoires de stockage.

6.8 Messages

Par message, on entend principalement une donnée utile, c'est-à-dire le résultat d'une écriture sur l'un des nœuds de communication du système de fichiers MPCFS. Toutefois, afin d'uniformiser les traitements, la structure que l'on utilise pour décrire et transmettre ces messages utiles est aussi partiellement utilisée pour la transmission des informations de contrôle liées aux protocoles de communication du système MPCFS.

Les informations contenues dans la structure de données associée aux messages sont de deux natures :

1. les informations ayant trait à la **description du message**. Les principaux champs que l'on trouve dans cette catégorie sont les suivants :
   - l'**identifiant local** du message, c'est-à-dire son numéro dans le répertoire de stockage où il est placé ;
   - la **date de création (ou d'arrivée)** du message ;
   - une référence à la structure associée à l'**expéditeur du message** : selon que le message est émis par un processus local ou reçu d'un processus distant, cette référence correspond à un membre d'un groupe de communication ou à une association.
   - une référence à la structure locale du **sous-groupe source ou destination** du message ;
   - une référence aux **zones mémoire contenant les données utiles** du message ;
   - la **longueur du message** ;
   - le **type du message** (donnée utile ou message de contrôle).
2. les informations de contrôle utilisée pour la **transmission du message** ;
   - une référence vers **la liste des associations destinataires du message** : cette liste doit être construite à chaque envoi de message pour faire face à la dynamicité des créations et destructions d'associations. Le message ne doit être envoyé qu'à destination des seules associations actives dans le groupe de communication au moment de la prise en charge du message ;
   - un compteur indiquant le nombre de **connexions ayant refusé le message par manque de place dans les files d'émission locales** ;
   - un compteur indiquant le nombre de **connexions ayant refusé le message par manque de place** sur les machines distantes ;
   - un **verrou** permettant d'assurer l'exclusion mutuelle des processus dans les sections critiques ;
   - des **drapeaux** liés au protocole de communication ;
   - un **code d'erreur**, positionné lorsque le message est une réponse à un message précédent ;
   - des **compteurs de référence** au message.

6.9 Conclusion

Dans ce chapitre, nous avons présenté les principales structures de données utilisées dans notre prototype du système de fichiers MPCFS.

Cette description nous a permis d'introduire certains des problèmes de gestion liés aux groupes de communication, que nous allons aborder de façon plus détaillée dans le chapitre suivant. Au travers des structures de connexion, d'association et de message, elle nous a aussi permis d'entrevoir les difficultés de conception et de mise en œuvre des couches de protocole, auxquelles nous nous sommes heurtés lors du développement de ce prototype MPCFS. Nous revenons plus en détail sur les algorithmes liés à la gestion des communications dans la deuxième partie du prochain chapitre, qui est consacrée à la partie basse du système. Enfin, nous terminons cette description du prototype par la présentation des protocoles de communication proprement dit, au chapitre 8.

Chapitre 7

Algorithmes

7.1 Introduction

Dans ce chapitre, nous présentons les principaux algorithmes que nous avons conçus et mis en œuvre dans le prototype de MPCFS. Notre objectif ici est avant tout de donner une vue d'ensemble du fonctionnement du système, de souligner les difficultés algorithmiques auxquelles nous nous sommes heurté lors de sa construction et, finalement, de donner une idée du travail de développement qu'implique son adaptation à une plate-forme UNIX quelconque. Dans la présentation de ces algorithmes, nous avons ainsi cherché à nous abstraire des considérations typiquement liées à la plate-forme cible (en l'occurrence ici, le système Linux).

Cette présentation se décompose en trois parties :

- l'architecture générale du système est présentée au paragraphe 7.2 ;
- les principaux algorithmes de la partie haute du système, c'est-à-dire situés à l'interface entre les programmes des utilisateurs et le système MPCFS, sont présentés au paragraphe 7.3 ;
- les principaux algorithmes de la partie basse du système, concernant la gestion des opérations d'entrées/sorties sur le réseau, sont présentés au paragraphe 7.4 ;

7.2 Fonctionnement général

La figure 7.1 présente l'architecture logicielle du système MPCFS. Avant de décrire chacun des principaux algorithmes qui constituent le système MPCFS, il convient de s'attarder quelque peu sur les principes généraux de fonctionnement de ce système, et en particulier sur l'architecture à fils d'exécution multiples permettant l'exécution de ces algorithmes. En effet, la programmation au niveau du noyau d'un système UNIX s'accompagne de quelques contraintes, qui viennent s'ajouter aux difficultés intrinsèques des algorithmes.

Les deux principales contraintes auxquelles nous avons du faire face sont les suivantes :

- l'impossibilité pour le noyau d'accéder à des fonctionnalités auxquelles tout processus qui s'exécute en contexte utilisateur a pourtant librement accès ;
- la nécessité de réaliser des traitements asynchrones, c'est-à-dire en arrière plan, en dehors du contexte d'exécution des processus clients du système.

Par exemple, la création de *sockets* ou l'accès à un service de conversion des noms de machines en adresse IP (DNS) sont des fonctionnalités dont nous avions impérativement besoin, mais qui sont inaccessibles à partir du contexte d'exécution du noyau Linux. Ces fonctionnalités sont indispensables pour le traitement des requêtes d'association entre les groupes de communication locaux et ceux des autres machines. La création de *sockets* est nécessaire afin d'établir une connexion point-à-point avec chacune des machines et, la conversion d'adresses, afin d'éviter toute ambiguïté, lorsque plusieurs noms sont associés à l'adresse IP d'une machine.

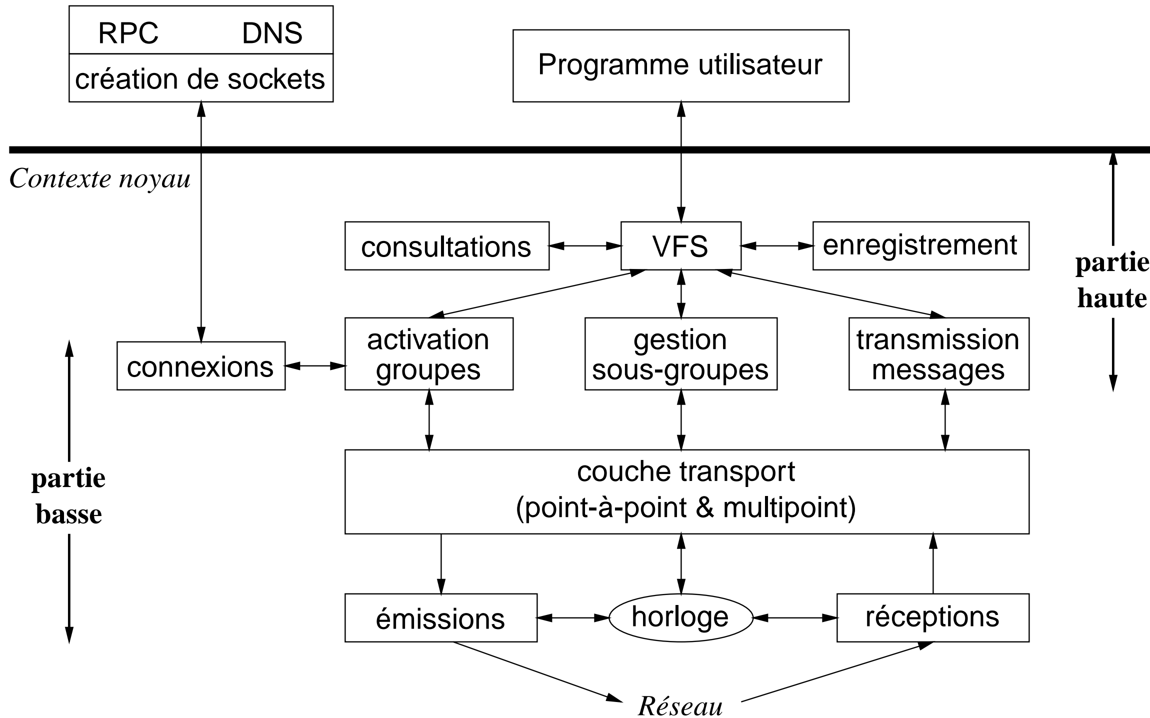


FIG. 7.1: *Architecture logicielle du système MPCFS*

Quant à la nécessité de réaliser des traitements asynchrones, elle apparaît au niveau des échanges de messages. Par exemple, le système ne peut pas attendre qu'un processus utilisateur déclenche une opération sur le système de fichiers pour traiter les messages reçus des autres machines. Rappelons en effet, que les algorithmes du noyau sont normalement déclenchés à la suite d'un appel système ou lors du traitement d'une interruption. Le recours à cette dernière solution doit toutefois être évité, car le contexte d'exécution des interruptions impose des contraintes sévères sur le temps de réalisation des algorithmes et sur les fonctionnalités auxquels ils ont accès (l'allocation mémoire, notamment).

Pour faire face à ces deux problèmes, nous avons attaché au noyau initial deux processus de service principaux : l'un chargé de réaliser en contexte d'exécution utilisateur les opérations qui sont indisponibles dans le contexte d'exécution du noyau, et l'autre chargé de réaliser les traitements asynchrones.

7.2.1 Processus de services

L'accès à l'ensemble des fonctionnalités qui ne sont pas disponibles dans le contexte du noyau, mais dont a besoin le système MPCFS, est fourni par un processus de service appelé `connd`. Ces fonctionnalités ne sont en fait requises que lors des phases d'association des groupes de communication entre machines.

Ce processus alterne son exécution entre le contexte du noyau et le contexte utilisateur,

au travers d'appels systèmes surchargés grâce au mécanisme de *VFS* [Card 93]. Ce processus est généralement inactif (endormi). Il n'a pour seule fonction que le traitement des trois types de requêtes suivantes, qui exigent chacune un retour en contexte d'exécution utilisateur :

– les requêtes de connexion émanant des processus locaux ;

– les requêtes de connexion de type RPC émanant des processus distants ;

– les requêtes de vérification de la validité d'une demande d'association de groupe de communication entre la machine locale et une machine distante ;

En ce qui concerne les connexions, nous nous sommes inspiré dans MPCFS du protocole NFS [Sandberg 85, Sun Microsystems, Inc. 89, Callaghan 95]. Dans NFS, l'établissement d'une connexion s'effectue lors d'une opération de montage, au travers d'un appel RPC [Sun Microsystems, Inc. 90].

Au niveau du serveur, ce service RPC de montage NFS est assuré (dans Linux) par un processus de service indépendant, s'exécutant en contexte utilisateur. Ce processus est notamment chargé d'établir les permissions qui peuvent être accordées au client, d'après la configuration établie par l'administrateur du système dans un fichier de configuration (`/etc/exports` pour Linux). En cas de succès, le serveur retourne un identificateur de partition au client, que ce dernier doit ensuite fournir au serveur dans chacune de ses transactions.

Au niveau du client, le service RPC de montage NFS est invoqué directement par le processus exécutant la commande `mount()`. Brièvement, cette commande est chargée de récupérer l'identificateur de partition auprès du serveur et d'initialiser le *socket* de travail pour la partie cliente qui elle, s'exécute dans le contexte du noyau. Ces deux données sont transmises du contexte d'exécution utilisateur (celui de la commande `mount()`) vers le contexte d'exécution du noyau au travers de l'appel système `sys_mount()`.

De façon similaire, la gestion des connexions du système MPCFS est assurée au travers d'un service RPC, qui est pris en charge en contexte d'exécution utilisateur par un processus de service. Notons toutefois que dans MPCFS, c'est le même processus de service qui prend en charge l'exécution de la partie cliente et de la partie serveur. La transmission dans le contexte d'exécution du noyau des paramètres de la connexion obtenus par ce processus de service se fait au travers d'appels systèmes. Pour cela, l'arborescence MPCFS contient des fichiers, dont l'utilisation est limitée aux processus de service. Certains des appels systèmes qui concernent ces fichiers (`sys_open()`, `sys_read()`, `sys_write()`, `sys_select()` et `sys_close()`) ont été surchargés, grâce au mécanisme de *VFS*.

Les traitements en tâche de fond sont assurés par un second processus de service, appelé **`sockd`**. Ce processus assure la mise en œuvre des couches de protocole des niveaux transport et session de MPCFS, à partir du contexte d'exécution du noyau. Ce processus ne retourne jamais en contexte d'exécution utilisateur.

De plus, nous avons été amenés à mettre en place des processus de service secondaires, appelés **`sockiod`** et chargés d'assister le processus **`sockd`**. Le processus **`sockd`** fait appel à

l'un de ces processus lorsqu'il n'a d'autre possibilité que de recourir à une opération d'entrée/sortie bloquante. En effet, la gestion des couches de protocole dont est chargé le processus **sockd** exige de ce dernier une disponibilité de tous les instants. Plusieurs instances du processus **sockiod** peuvent être lancées (l'actuel prototype en utilise quatre). Avec l'actuel prototype pour le système Linux, la seule opération bloquante dont se décharge le processus **sockd** auprès de ces processus assistants est la fonction d'émission bloquante d'un message sur un *socket* [1]. Ces processus de service secondaires ne quittent jamais le contexte d'exécution du noyau.

7.2.2 L'architecture à fils d'exécution multiple du prototype MPCFS

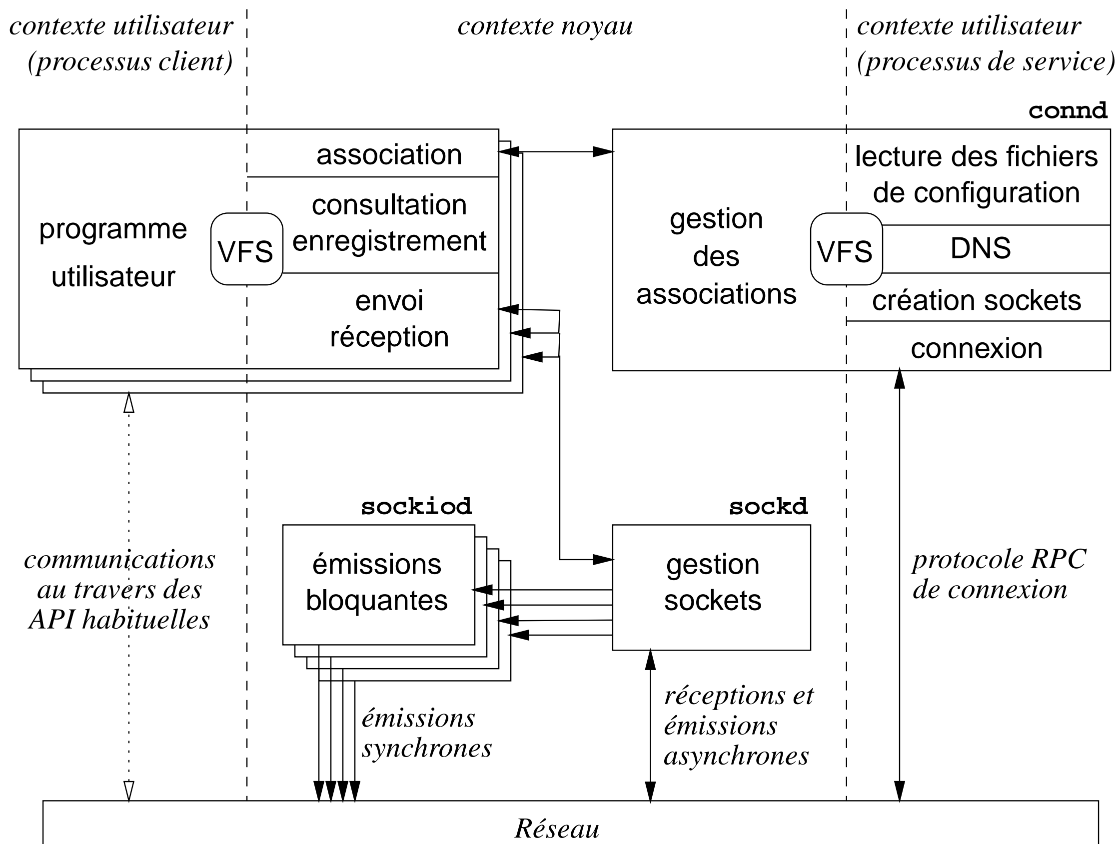


FIG. 7.2: *Architecture à fils d'exécution multiples du prototype MPCFS*

L'utilisation de ces processus de service aboutit à l'architecture à fils d'exécution multiples représentée par la figure 7.2. Les processus clients, c'est-à-dire ceux qui exécutent les programmes des utilisateurs, exploitent les fonctionnalités du système MPCFS par l'intermédiaire d'appels au système *surchargés* grâce au mécanisme de *VFS*.

1. Par défaut, **sockd** cherche d'abord à envoyer ses messages en utilisant une fonction d'émission non-bloquante, dont le succès n'est hélas pas systématiquement garanti.

Bien que le noyau Linux soit écrit en C, il s'appuie sur des techniques de programmation objet, et en particulier celle des appels de méthode surchargés. Dans le cas de Linux, la surcharge consiste à permettre l'ajout de traitements spécifiques, à chacun des traitements associés par défaut à un objet. Par exemple, supposons qu'un pilote de périphérique désire afficher l'heure de chaque ouverture d'un fichier dont il a la charge. Pour cela, il lui suffit d'ajouter à la méthode d'ouverture par défaut (déclenchée lors de l'appel système `sys_open()`), sa propre méthode, réalisant l'affichage de l'heure.

En ce qui concerne MPCFS, les deux principaux "objets" que nous avons surchargé sont les i-nœuds et les fichiers. La surcharge des méthodes concernant les i-nœuds est réalisée de façon dynamique, lorsque le système demande au pilote MPCFS de remplir la structure d'un i-nœud (au travers de la fonction `read_inode()`, que nous avons déjà évoquée au paragraphe 6.2, page 142). Quant à la surcharge des méthodes concernant les fichiers, elle est simplement attachée à celle concernant les i-nœuds, de façon statique. Soulignons en particulier que cette surcharge dynamique des méthodes, lors de la récupération des i-nœuds, permet d'associer un traitement différent à *chacune des opérations* concernant *chacun des fichiers et répertoires* d'une arborescence.

Les traitements du système de fichiers MPCFS sont donc déclenchés automatiquement par le mécanisme de VFS, en fonction des opérations surchargées et des fichiers concernés. Certains de ces traitements, comme les requêtes de consultation du système de fichiers, ou les requêtes d'enregistrement dans un groupe de communication, n'exigent pas d'interaction avec les processus de service ou les autres processus utilisateurs. D'autres traitements peuvent simplement impliquer des phases de synchronisation avec les autres processus utilisateurs. Ils ont pour cela recours au mécanisme des files d'attente décrit au paragraphe [Rubini 98]. Mais dans la plupart des cas, ces traitements font appel aux processus de service : **`connd`** s'il s'agit d'une requête d'association de groupes de communication, **`sockd`** sinon.

7.2.3 Fonctionnement en mode restreint

En l'absence de processus de service, le système MPCFS fonctionne en mode restreint. Dans ce mode de fonctionnement, le mécanisme d'association est indisponible et, par conséquent, toutes les opérations impliquant l'utilisation du réseau sont impossibles. En revanche, le système reste parfaitement opérationnel pour toute autre opération, que ce soit l'enregistrement d'un processus dans un groupe de communication, la communication inter-processus, les créations et destructions de sous-groupes, ou encore la consultation du système de fichiers.

Pour faire passer le système du mode de fonctionnement restreint au mode de fonctionnement nominal, il suffit de lancer l'exécution du processus **`sockd`**, lequel lance à son tour l'exécution des processus **`connd`** et **`sockiod`**.

7.3 Algorithmes de la partie haute

Nous allons maintenant présenter les principaux algorithmes de la partie haute du système. Ces algorithmes sont exécutés par le système MPCFS lorsqu'un processus utilisateur déclenche l'un des appels au système que nous avons surchargé au moyen du mécanisme de *VFS*.

Plus précisément, les algorithmes que nous allons présenter sont les suivants :

- l'algorithme d'**enregistrement d'un processus dans un groupe de communication**, déclenché par une opération d'écriture sur le fichier `register` (voir paragraphe 5.3.2.9, page 123) ;
- l'algorithme de **dés-enregistrement d'un processus d'un groupe de communication**, déclenché par une opération d'écriture sur le même fichier `register` ;
- l'algorithme d'**émission d'un message**, déclenché par une opération d'écriture sur l'un des nœuds de communication de l'arborescence MPCFS (voir paragraphe 5.3.2.5, page 119) ;
- l'algorithme de **réception d'un message**, déclenché par une opération d'écriture sur l'un des nœuds de communication de l'arborescence MPCFS ;
- l'algorithme d'**affiliation d'un processus membre à un sous-groupe de communication**, déclenché par une opération de création de répertoire dans l'un des répertoires de l'arborescence propre d'un processus membre (voir paragraphe 5.3.2.3, page 116) ;
- l'algorithme de **dés-affiliation d'un processus membre d'un sous-groupe de communication**, déclenché par une opération de destruction de répertoire dans l'un des répertoires de l'arborescence propre d'un processus membre.

7.3.1 Algorithme d'enregistrement

Lorsqu'un processus veut communiquer avec d'autres processus au travers d'un groupe de communication, soit il utilise les nœuds de communications d'autres processus déjà enregistrés dans ce groupe, soit il s'enregistre lui-même dans ce groupe pour construire (éventuellement) sa propre arborescence de multiplexage. S'il décide de s'enregistrer dans le groupe, il doit écrire le nom de ce groupe et les paramètres de sa demande dans le fichier d'enregistrement `register`. Nous allons maintenant décrire l'algorithme déclenché lors de cette demande d'enregistrement.

Algorithme principal L'algorithme 7.1 commence par vérifier si le groupe existe déjà. Si c'est le cas, il vérifie si le processus qui désire s'y enregistrer n'en fait pas déjà partie. S'il en fait déjà partie la requête échoue. Si le groupe n'existe pas déjà, il essaie de le créer (algorithme présenté au paragraphe suivant) ; l'algorithme échoue si cette création échoue. Si le groupe existe déjà, l'algorithme vérifie que son nombre maximum de membres n'a pas

ALG. 7.1 Enregistrement dans un groupe de communication

```
fonction enregistrer_groupe(nom_groupe,ident_processus,paramètres)
début
  grp←retrouver_groupe(nom_groupe)

  si (grp existe) alors
    mmbr←retrouver_membre(grp,ident_processus)
    si (mmbr existe) alors retour erreur:DÉJÀ_MEMBRE
    sinon
      si (grp→nombre_membres = MAX_MEMBRES_PAR_GROUPE) alors
        retour erreur:MAX_MEMBRES_ATTEINT finsi

      perm←permissions_utilisation_groupe(grp,ident_processus,paramètres)
      si non (perm) alors retour erreur:PERMISSION_REFUSÉE finsi
    finsi
  sinon
    val←créer_groupe(nom_groupe,ident_processus,paramètres)
    si (val ≠ OK) alors retour val
    sinon grp←retrouver_groupe(nom_groupe) finsi
  finsi

  mmbr←initialiser_structure_membre(grp,ident_processus,paramètres)
  insérer_liste(grp→membres,mmbr)
  incrémenter(grp→compteur_membres)
  sgrp←premier_liste(grp→sous_groupes)                    {sous-groupe racine}
  insérer_liste(mmbr→sous_groupes,sgrp)
  incrémenter(sgrp→compteur_membres)
  incrémenter(mmbr→compteur_sous_groupes)

  retour OK
fin

fonction créer_groupe(nom_groupe,ident_processus,paramètres)
début
  perm←permissions_création_groupe(nom_groupe,ident_processus,paramètres)
  si non (perm) alors retour erreur:PERMISSION_REFUSÉE finsi

  grpid←attribuer_identificateur_groupe()
  si non (grpid valide) alors retour erreur:MAX_GROUPES_ATTEINT finsi

  grp←initialiser_structure_groupe(grpid,nom_groupe,ident_processus,paramètres)
  table_groupes[grpid]←grp
  sgrp←créer_sous_groupe_racine(grp)
  insérer_liste(grp→sous_groupes,sgrp)

  retour OK
fin
```

été atteint et que le processus demandeur a bien la permission d'utiliser ce groupe. Dans le cas contraire, la requête échoue.

A ce point, si l'algorithme n'a pas encore échoué, il ne le peut plus : il crée la structure du nouveau membre, l'initialise à partir des paramètres de l'appel et l'insère dans la liste des membres du groupe. Finalement, il insère le sous-groupe racine du groupe dans la liste des sous-groupes du nouveau membre et incrémente les compteurs de références des différentes structures en conséquence.

Algorithme de création d'un groupe L'algorithme de création commence par vérifier que le processus demandeur a effectivement la permission de créer ce groupe avec les paramètres demandés (fonction permissions_création_groupe() dans l'algorithme 7.1), en consultant la configuration établie par l'administrateur du système dans le fichier `/etc/mpc_access`. Si le demandeur a effectivement la permission de créer le groupe, l'algorithme recherche le premier identificateur de groupe disponible (fonction attribuer_identificateur_groupe() dans l'algorithme 7.1). Si aucun identificateur n'est disponible, c'est que le nombre maximal de groupes est atteint et la demande échoue. Sinon, l'algorithme alloue une nouvelle structure de groupe et l'initialise à partir des paramètres de l'appel. Il crée ensuite le sous-groupe racine du groupe et l'insère dans la liste des sous-groupes du groupe.

7.3.2 Algorithme de dés-enregistrement

Algorithme principal L'algorithme 7.2 commence par vérifier que le groupe dont le nom lui a été passé en paramètre existe bien et que le processus qui demande à s'en dés-enregistrer en est bien membre. Dans le cas contraire, il échoue. Il vérifie ensuite que l'arborescence de sous-groupes de ce membre est bien réduite au seul sous-groupe racine. Dans le cas contraire, le dés-enregistrement est impossible et la demande échoue.

Il décrémente ensuite les compteurs de membres du groupe et de la racine de l'arborescence des sous-groupes du groupe. Puis il retire la structure de membre de la liste des membres du groupe afin d'en interdire l'accès au travers de l'arborescence MPCFS (le répertoire propre de ce membre n'apparaît plus dans la liste des fichiers du répertoire racine du groupe). Toutefois, la structure du membre n'est pas immédiatement détruite, si son compteur de référence est supérieur à zéro. En effet, d'autres processus peuvent être en train d'utiliser le répertoire propre de ce processus membre ou l'un des fichiers qui y apparaissent. Ce n'est que lorsque tous ces processus en auront terminé avec les fichiers de l'arborescence propre de ce membre que la structure pourra effectivement être détruite. Ce mode de fonctionnement respecte donc la sémantique habituelle d'une destruction de fichier dans les systèmes UNIX.

L'algorithme vérifie ensuite s'il reste des membres dans le groupe. Si ce n'est pas le cas et que le compteur des références au groupe est nul, le groupe peut être détruit. Si le compteur des références à la structure du groupe n'est pas nul, la structure du groupe ne peut être détruite. Elle reste toutefois accessible à partir du système de fichiers, ce qui diffère quelque peu de la sémantique habituelle d'une destruction, mais n'a que peu de conséquences pour l'utilisateur.

ALG. 7.2 Dés-enregistrement d'un groupe de communication

```
fonction dés-enregistrer_groupe(nom_groupe,ident_processus)
début
  grp←retrouver_groupe(nom_groupe)
  si non (grp existe) alors retour erreur:GROUPE_INEXISTANT finsi

  mmbr←retrouver_membre(grp,ident_processus)
  si non (mmbr existe) alors retour erreur:MEMBRE_NON_ENREGISTRÉ finsi

  si (mmbr→compteur_sous_groupes > 1) alors
    retour erreur:SOUS_GROUPES_NON_DÉTRUITS finsi

  sgrp←premier_liste(grp→sous_groupes)
  décrémenter(sgrp→compteur_membres)

  décrémenter(grp→compteur_membres)
  retirer_liste(grp→membres,mmbr)
  si (mmbr→compteur_références = 0) alors
    détruire_structure_membre(mmbr)
  finsi

  si ((grp→compteur_membres = 0) et (grp→compteur_références = 0)) alors
    détruire_groupe(grp)
  finsi

  retour OK
fin

fonction détruire_groupe(nom_groupe,ident_processus)
début
  sgrp←premier_liste(grp→sous_groupes)                    {sous-groupe racine}

  {puisque le compteur de références du groupe est nul,}
  {celui du sous-groupe racine l'est aussi.}
  détruire_structure_sous_groupe(sgrp)

  grpid←(grp→id)
  table_groupes[grpid]←0
  restituer_identificateur_groupe(grpid)
  détruire_structure_groupe(grp)
  retour OK
fin
```

Destruction d'un groupe La destruction de la structure du groupe de communication, quand elle est possible (et nécessaire), est réalisée de la façon suivante :

1. le sous-groupe racine du groupe est détruit. Il peut l'être car si un processus possédait une référence vers cette structure, il en posséderait alors obligatoirement une vers la structure du groupe, ce qui, en amont, aurait interdit cette destruction ;

2. l'identificateur du groupe est restitué afin d'être recyclé. Les identificateurs, qui sont aussi des index dans une table de taille fixe (voir paragraphe 6.3.2, page 145), ne sont en effet disponibles qu'en quantité limitée ;

3. la structure du groupe est détruite.

7.3.3 Émission d'un message

L'algorithme 7.3 est invoqué au travers de l'appel système `sys_write()`, appliqué à l'un des fichiers de type nœud de communication du système de fichiers MPCFS. En contexte d'exécution noyau, cet appel système fournit des références vers les structures du fichier et de l'i-nœud associé, en plus des paramètres de l'appel système fournis par l'utilisateur.

L'algorithme commence par récupérer les structures du groupe de communication, du sous-groupe et du membre du groupe correspondant au nœud de communication utilisé pour l'écriture. Il s'appuie pour cela sur la numérotation structurée des i-nœuds présentée à la section 6.2 (page 142). Il vérifie ensuite qu'il existe bien d'autres membres du groupe de communication susceptibles de recevoir ce message : si le processus membre par l'intermédiaire duquel le message est envoyé est le seul membre du groupe à définir le sous-groupe permettant de recevoir ce message, alors ce message n'a aucun destinataire et l'algorithme peut se terminer immédiatement, avec succès.

L'algorithme doit ensuite s'assurer que le système peut accueillir le message, c'est-à-dire que le nombre maximum de messages dans le sous-groupe n'a pas été atteint et que la quantité de mémoire déjà utilisée localement dans ce sous-groupe ainsi que globalement, pour tout le groupe de communication ne dépassent pas les maxima autorisés. Lorsque le système ne peut pas accueillir le message deux situations sont possibles. Soit le descripteur du fichier utilisé pour l'envoi est configuré pour opérer en mode non-bloquant et l'algorithme échoue en retournant une erreur ; soit le descripteur du fichier est configuré en mode bloquant et le processus est endormi jusqu'à ce que le message puisse enfin être accueilli.

Rappelons qu'en contexte d'exécution noyau, la méthode traditionnellement utilisée pour gérer la concurrence entre des processus qui désirent accéder à une même ressource consiste à définir une file d'attente pour cette ressource [Vahalia 96, Rubini 98]. Chaque processus ayant besoin de cette ressource s'insère dans la file d'attente correspondante et s'endort. Lorsqu'un processus libère la ressource attendue, il réveille tous les processus endormis sur cette file d'attente. Plusieurs processus peuvent alors se trouver à nouveau en compétition pour l'accès à cette ressource. Le conflit est géré de façon simple et équitable par l'ordonnanceur UNIX, qui choisit l'ordre d'exécution des processus en fonction de leur priorité. Les processus ont en effet une priorité d'autant plus élevée qu'ils attendent la ressource depuis

ALG. 7.3 Envoi d'un message

```
fonction envoyer_message(fichier, i-nœud, message, taille)
début
  grp←déduire_groupe(i-nœud)
  sgrp←déduire_sous_groupe(i-nœud)
  mmbr←déduire_membre(i-nœud)

  si (sgrp→compteur_membre ≤ 1) alors retour taille finsi

  tant que ((sgrp→nombre_messages = MAX_MESSAGE_PAR_SOUS_GROUPE) ou
           (taille > espace_disponible(grp,sgrp)))
  faire
    si (fichier→mode_non_bloquant = vrai) alors
      retour erreur:OPÉRATION_BLOQUERAIT finsi

    dormir_sur_file_attente(sgrp→écriture_bloquée)
  fait

  msg←(sgrp→sentinelle)
  créer_nouvelle_sentinelle(sgrp)
  copie_corps_message(de:message vers:msg)

  initialiser_structure_message(message,taille,grp,sgrp,mmbr)
  (msg→nombre_destinataires)←(sgrp→compteur_membres − 1)
  (msg→identifiant)←attribuer_identifiant_message(sgrp)

  mettre_à_jour_statistiques_écriture(grp,sgrp,taille)
  réveiller_file_attente(sgrp→lecture_bloquée)
  diffuser_message(msg,sgrp,mmbr)

  retour taille
fin
```

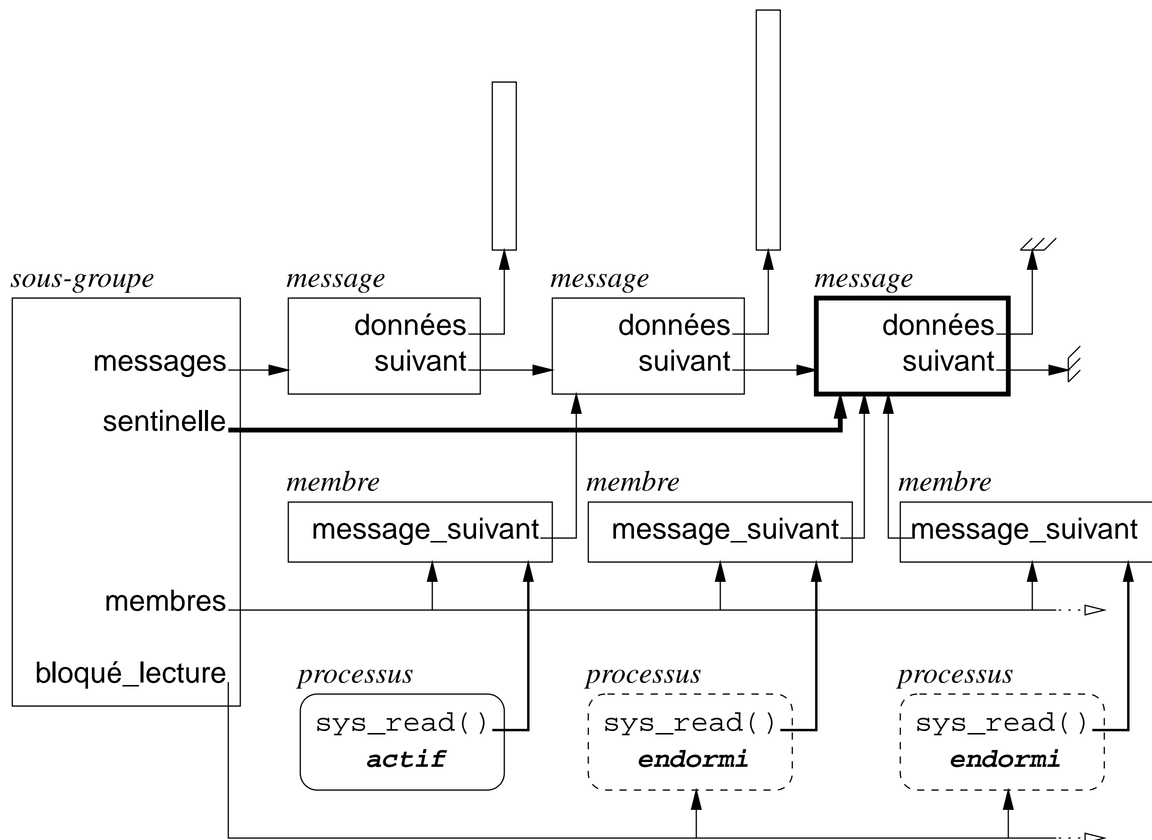


FIG. 7.3: *Structure de message sentinelle utilisée dans les sous-groupes*

longtemps[2]. Les premiers processus à reprendre leur exécution (c'est-à-dire ceux qui ont la priorité la plus haute) trouvent la ressource disponible et se l'approprient. Les autres doivent se rendormir après s'être replacé sur la file d'attente. En pratique, cette façon de procéder exige donc d'un processus qu'il vérifie systématiquement que la ressource attendue est bien disponible lorsqu'il se réveille.

Signalons toutefois que certains systèmes UNIX (Solaris, notamment), permettent de ne demander que le réveil du processus ayant la plus grande priorité. Dans ce cas, la boucle est inutile.

Cette étape franchie, l'algorithme fait une copie dans la mémoire du noyau de la zone de mémoire utilisateur contenant le corps du message puis initialise la structure décrivant le message (décrite au paragraphe 6.8, page 158). L'algorithme utilise pour cela une structure de message sentinelle, allouée lors d'un précédent envoi de message dans le sous-groupe. Cette structure permet à un processus bloqué dans l'attente d'un message à lire, de faire référence à la structure du prochain message, avant même que celui-ci ne soit disponible (figure 7.3).

2. Plus précisément la priorité des processus s'accroît ou diminue progressivement, selon qu'ils sont endormis ou qu'ils s'exécutent. Les processus qui sont endormis dans l'attente d'une ressource voient donc leur priorité augmenter, alors que ceux qui l'obtiennent voient la leur diminuer.

L'algorithme réveille ensuite les processus bloqués dans l'attente d'un message à lire dans le sous-groupe. Finalement, le message est diffusé vers les machines associées au groupe qui ont défini ce sous-groupe.

7.3.4 Réception d'un message

De la même façon que pour l'algorithme d'émission, l'algorithme 7.4 commence par récupérer les structures du groupe de communication, du sous-groupe et du membre du groupe correspondant au nœud de communication utilisé pour la lecture.

Il vérifie ensuite qu'il a bien un message à lire par l'intermédiaire de ce nœud de communication. Pour cela, il vérifie que la référence au prochain message à lire dans ce sous-groupe n'est pas celle de la structure sentinelle du sous-groupe. Dans le cas où cette référence est celle de la structure sentinelle, selon que le fichier utilisé pour la lecture est configuré en mode bloquant ou non, l'algorithme retourne une erreur ou bien s'endort après s'être placé sur la file d'attente des processus bloqués en lecture. Les processus endormis sur cette file d'attente ne sont par la suite réveillés que lorsqu'un processus envoie un message dans le sous-groupe ou qu'un message en provenance d'une autre machine est reçu (ou que le processus reçoit un signal non masqué).

Lorsque le processus est réveillé, il s'assure systématiquement qu'il a effectivement un message à lire. L'algorithme doit pour cela ignorer les messages qui ont été envoyés par l'intermédiaire du nœud de communication au travers duquel la lecture est en train de s'opérer (fonction message_suivant()). En effet, rappelons qu'un nœud de communication peut être simultanément utilisé par plusieurs processus, tant en lecture qu'en écriture.

Lorsqu'un message est trouvé, l'algorithme peut le copier dans le contexte d'exécution du processus appelant et mettre à jour les statistiques d'utilisation du groupe et du sous-groupe.

Ensuite, l'algorithme décrémente le nombre de destinataires de ce message. Si ce nombre tombe à zéro, c'est que le message a été consommé par tous ses destinataires. Dans ce cas, le message est retiré de la liste des messages du sous-groupe et l'identifiant du message est restitué. De plus, les processus éventuellement bloqués en écriture dans ce sous-groupe sont réveillés. Par conséquent, le fichier correspondant à ce message dans l'arborescence de stockage devient inaccessible à d'autres processus que ceux qui l'avaient ouvert avant cette lecture. Si aucun autre processus n'a ouvert ce fichier, c'est-à-dire si son compteur de référence est nul, le message peut être détruit. Dans le cas contraire, le message ne sera effectivement détruit que lorsque le dernier des processus ayant ouvert le fichier avant cette lecture l'aura refermé.

7.3.5 Affiliation d'un processus membre à un sous-groupe

L'algorithme 7.5 est invoqué par une opération de création de répertoire (`mkdir()`) dans l'arborescence des sous-groupes d'un processus membre. En contexte d'exécution noyau, cet appel système fournit une référence à la structure de l'i-nœud du répertoire parent et le nom du répertoire fils que l'utilisateur désire créer.

ALG. 7.4 Réception d'un message

```
fonction recevoir_message(fichier, i-nœud, message, taille)
début
  grp←déduire_groupe(i-nœud)
  sgrp←déduire_sous_groupe(i-nœud)
  mmbr←déduire_membre(i-nœud)

  tant que (message_suivant(sgrp,mmbr) = sgrp→sentinelle)
  faire
    si (fichier→mode_non_bloquant = vrai) alors
      retour erreur:OPÉRATION_BLOQUERAIT finsi

    dormir_sur_file_attente(sgrp→lecture_bloquée)
  fait

  msg←message_suivant(sgrp,mmbr)
  taille_réelle←min(taille,msg→taille)
  copie_corps_message(de:msg vers:message)
  mettre_à_jour_statistiques_lecture(grp,sgrp,taille_réelle)

  décrémenter(msg→nombre_destinataires)
  si (msg→compteur_lectures = 0) alors
    retirer_liste(sgrp→messages,msg)
    restituer_identifiant_message(msg)
    réveiller_file_attente(sgrp→écriture_bloquée)
    si (msg→compteur_réferences = 0) alors
      détruire_message(msg)
    finsi
  finsi

  retour taille_réelle
fin

fonction message_suivant(sgrp,mmbr)
début
  tant que ( (mmbr→message_suivant ≠ sgrp→sentinelle) et
             (mmbr→message_suivant→expéditeur = mmbr))
  faire
    {Ignorer les messages dont on est l'expéditeur !}
    mmbr→message_suivant←suivant_liste(mmbr→message_suivant)
  fait

  retour mmbr→message_suivant
fin
```

ALG. 7.5 Affiliation à un sous-groupe

```
fonction affilier_sous_groupe(i-nœud, nom)
début
  grp←déduire_groupe(i-nœud)
  sgrp_parent←déduire_sous_groupe(i-nœud)
  mmbr←déduire_membre(i-nœud)

  sgrp←rechercher_répertoire_multiplexage(mmbr→sous_groupes,sgrp_parent,nom)

  si (sgrp existe) alors
    retour erreur:DÉJÀ_AFFILIÉ
  sinon
    sgrp←rechercher_sous_groupe(sgrp_parent→fils,sgrp_parent,nom)

    si non (sgrp existe) alors
      sgrp←créer_sous_groupe(grp,sgrp_parent,nom)
  finsi

  ajouter_liste_sous_groupe(mmbr,sgrp)
  initialiser_sous_groupe_membre(mmbr,sgrp)

  retour OK
fin

fonction créer_sous_groupe(grp, sgrp_parent, nom)
début
  sgrp←initialiser_structure_sous_groupe(grp,sgrp_parent,nom)
  ajouter_liste_sous_groupe(grp,sgrp)
  insérer_arborescence(grp,sgrp)
  incrémenter(grp→compteur_sous_groupes)

  si (sgrp_parent→référence_autres_machines > 0) alors
    annoncer_nouveau_sous_groupe(sgrp)
  finsi

  retour sgrp
fin
```

La première étape de l'algorithme consiste donc à récupérer les structures du groupe de communication, du sous-groupe et du membre du groupe, à partir du numéro de l'i-nœud. L'algorithme vérifie ensuite que le répertoire dont le processus demande la création n'existe pas déjà, ce qui signifierait que le processus membre du groupe est déjà affilié à ce sous-groupe. Cette vérification est faite en parcourant la liste des répertoires de multiplexage que le processus membre a créé. L'algorithme recherche dans cette liste un répertoire portant le nom reçu en paramètre de l'algorithme et ayant pour parent le répertoire correspondant à l'i-nœud reçu en paramètre. Si un tel répertoire existe déjà, la demande échoue.

Si le répertoire demandé n'existe pas, l'algorithme vérifie si le sous-groupe correspondant existe déjà dans le groupe de communication, pour un autre processus membre. Cette vérification est faite en parcourant la liste des sous-groupes fils, à la recherche d'un sous-groupe portant le nom reçu en paramètre de l'algorithme. Si le sous-groupe n'existe pas déjà, l'algorithme doit le créer.

La création d'un sous-groupe est réalisée en deux temps. Tout d'abord la structure du sous groupe est créée, initialisée et insérée dans l'arborescence des sous-groupes du groupe de communication. Cette création entraîne l'apparition d'un répertoire de stockage pour ce sous-groupe dans l'arborescence de stockage des messages. Dans un deuxième temps, l'algorithme de création avertit les autres machines de l'existence de ce sous-groupe (à l'aide du protocole décrit au paragraphe 8.3.4, page 211).

Enfin, l'algorithme peut ajouter le sous-groupe à la liste des sous-groupes du processus membre et initialiser la structure correspondante pour ce membre, notamment en plaçant la référence au prochain message à lire dans ce sous-groupe, sur la structure de message sentinelle du sous-groupe (les messages éventuellement reçus dans ce sous-groupe avant l'affiliation sont ignorés).

7.3.6 Dés-affiliation d'un processus membre d'un sous-groupe

De la même façon que l'algorithme d'affiliation, l'algorithme 7.6 commence par récupérer les structures du groupe de communication du sous-groupe et du membre du groupe à partir du numéro de l'i-nœud du répertoire parent passé en paramètre. Il recherche ensuite le répertoire dont le nom est reçu en paramètre dans la liste des répertoires de multiplexage du processus membre. Si le répertoire n'existe pas, c'est que le membre n'est pas affilié au sous-groupe correspondant et la demande échoue.

Puis, il consomme chacun des messages éventuellement en attente pour ce processus membre dans ce sous-groupe. Cette opération s'effectue de façon similaire à la lecture d'un message (algorithme 7.4), à ceci près que le message n'est pas copié dans l'espace mémoire utilisateur du processus appelant.

Remarquons que cette consommation correspond en fait exactement au résultat que produit une opération d'interdiction en lecture à tous, du nœud de communication du processus membre dans le sous-groupe (appel système `chmod()`).

Dès lors que l'algorithme s'est assuré que le processus membre désirant se dés-affilier n'avait plus de message en attente dans ce sous-groupe, il peut retirer ce dernier de la liste

ALG. 7.6 Dés-affiliation d'un sous-groupe

```
fonction dés-affilier_sous_groupe(i-nœud, nom)
début
   grp←déduire_groupe(i-nœud)
   sgrp_parent←déduire_sous_groupe(i-nœud)
   mmbr←déduire_membre(i-nœud)

   sgrp←rechercher_répertoire_multiplexage(mmbr→sous_groupes,sgrp_parent,nom)

   si non (sgrp existe) alors
      retour erreur:NON_AFFILIÉ
   finsi

   tant que (message_suivant(sgrp,mmbr) ≠ sgrp→sentinelle)
   faire
      consommer_message_suivant(sgrp,mmbr)
   fait

   retirer_liste_sous_groupe(mmbr,sgrp)
   décrémenter(sgrp→compteur_membres)

   si (sgrp→compteur_membres = 0) alors
      retirer_arborescence(grp,sgrp)
      si (sgrp→compteur_références = 0) alors
         détruire_sous_groupe(sgrp)
      finsi
   finsi

   retour OK
fin
```

des sous-groupes de ce processus et décrémenter le compteur de membres affiliés du sous-groupe.

Si ce compteur tombe à zéro, le sous-groupe est supprimé de l'arborescence. Si son compteur de références est nul lui aussi, la structure du sous-groupe est définitivement détruite.

7.4 Algorithmes de la partie basse

Nous allons maintenant décrire les principaux algorithmes de la partie basse du système MPCFS. Ces algorithmes sont essentiellement chargés de l'interface avec le réseau et les couches de protocole du système MPCFS (ces couches de protocole sont décrites au chapitre 8).

Plus précisément, les algorithmes auxquels nous allons nous intéresser sont les suivants :

- **la boucle de gestion des *sockets*** est l'algorithme de plus haut niveau, chargé de coordonner les opérations réseau du système MPCFS. C'est cet algorithme qui est exécuté par le processus `sockd` lorsqu'il entre dans le contexte d'exécution du noyau ;
- l'algorithme de **traitement d'une connexion active**, invoqué par l'algorithme précédent lorsqu'une opération concernant l'une des connexions établies avec les autres machines est necéssaire ;
- l'**algorithme d'association**, invoqué lorsqu'un groupe de communication local est associé avec celui d'une autre machine ;
- le protocole de **dialogue entre les processus utilisateur et le processus `connd`** (déclenché lors d'une opération d'association) ;
- l'algorithme d'**établissement d'une connexion** avec une autre machine.

7.4.1 Gestion des *sockets*

De façon interne, MPCFS réalise ses communications au travers de l'API des *sockets*. Une connexion point-à-point est établie avec chacune des machines avec lesquelles une association de groupe est établie (si plusieurs associations existent entre deux machines, seule une connexion est utilisée). En effet, indépendamment de la méthode choisie pour réaliser la diffusion des messages (diffusion réelle ou *multi-unicast*), le système utilise des communications point-à-point afin de supporter la majeure partie de son trafic de contrôle avec chacune des autres machines. Les envois et réceptions d'accusés de réception, les retransmissions de messages perdus ou la mise en place des associations de groupes, par exemple, sont systématiquement réalisés au travers de communications point-à-point.

En réception, le prototype, qui s'appuie actuellement sur le protocole UDP, utilise un numéro de port UDP différent pour chacune de ces connexions point-à-point. Chacun de ces numéros de port implique donc l'utilisation d'un *socket* différent. Cette stratégie est certes

plus complexe à mettre en œuvre qu'une stratégie basée sur l'utilisation d'un unique numéro de port et donc d'un unique *socket* pour toutes les connexions, mais elle présente des avantages. Elle permet notamment au système de mieux résister aux pics de trafic émanant d'une machine. La capacité de stockage des messages attendant d'être lus sur un *socket* est en effet limitée par le noyau à 64 kilo-octets. Lorsque cette capacité est dépassée, les messages sont rejetés et doivent être retransmis. Avec cette stratégie multi-ports, la capacité de stockage augmente donc en proportion avec le nombre de connexions au lieu de rester constante. Mais surtout, seuls les messages émanants des machines ayant provoqué un dépassement de capacité sont rejetés. Ceci est d'une part, plus équitable et, d'autre part, permet la mise en place d'un mécanisme de contrôle de flux efficace. En effet, une machine dont les messages se perdent, peut présumer à juste titre qu'elle transmet trop vite et donc ajuster son débit. Dans le cas contraire, une machine peut perdre des messages par la faute d'une autre machine avec laquelle elle se trouve en concurrence, et la mise en place d'un mécanisme de contrôle de flux efficace devient très difficile.

La prise en charge de ces multiples *socket* est assurée par le processus **`sockd`**, à partir du contexte d'exécution du noyau. Ce processus entre dans le contexte d'exécution du noyau au travers d'un appel système, traduisant le passage du système du mode de fonctionnement restreint au mode de fonctionnement réseau. Cet appel système, `sys_write()`, est déclenché sur un fichier de l'arborescence MPCFS prévu à cet effet. Ce fichier n'est accessible qu'à un processus lancé par le super-utilisateur du système. Il est situé à la racine de l'arborescence et s'appelle `.sockd_req`. Cet appel système est intéressant, car il permet au processus **`sockd`** de se transmettre des informations depuis son contexte d'exécution utilisateur vers son contexte d'exécution noyau, lorsqu'il débute son exécution.

L'algorithme de gestion des *sockets* est chargé de parcourir chacune des connexions actives à la recherche de traitements à effectuer. Ces traitements sont de plusieurs types : traitement d'un évènement d'entrée/sortie sur le *socket* de la connexion, traitement d'une action programmée à la date courante, ou traitement d'une opération longue, différée lors d'une itération précédente.

Afin d'être équitable, tant vis-à-vis des autres processus du système que pour chacune des connexions actives, l'algorithme s'autorégule en ne fournissant qu'une quantité de travail limitée (mais raisonnable) à chaque itération, pour chacune des connexions actives (la politique d'autorégulation est présentée et discutée au paragraphe 7.4.2).

Enfin, lorsque toutes les connexions ont été traitées, l'algorithme s'endort soit dans l'attente d'un nouvel événement d'entrée/sortie, soit dans l'attente de l'échéance suivante d'une date de travail. Notons que lorsque l'algorithme estime avoir suffisamment travaillé pour l'itération courante (du fait de la politique d'autorégulation) et qu'il lui reste des traitements à réaliser, il ne s'endort pas après s'être placé sur une file d'attente. Il indique simplement à l'ordonnanceur du système qu'il peut être preempté par un processus de plus grande priorité, en restant sur la file des processus prêts.

7.4.2 Traitement d'une connexion active

Comme l'illustre la figure 7.2, page 165, la gestion des associations, la gestion des sous-groupes et la transmission de messages utilisateurs dans les groupes de communication, sont assurées d'une part dans la partie haute de MPCFS, dans le contexte d'exécution des processus utilisateurs et, d'autre part, dans la partie basse, par le processus **`sockd`**.

Ce découpage s'explique tant pour des raisons pratiques que pour des raisons de performance. Dans le sens descendant, les messages qui sont le résultat d'un appel système d'un processus utilisateur, sont préparés directement par ce processus avant d'être confiés à la partie basse du système qui peut les prendre en charge au niveau de la couche transport. Sur le plan des performances, le gain est appréciable lors des transmissions de messages utilisateurs en boucle locale (entre processus d'une même machine). Dès la fin de l'exécution de l'appel système `write()` par le processus expéditeur, le message est disponible pour ses destinataires locaux, sans qu'il soit besoin de recourir aux services d'un processus intermédiaire. Sur le plan pratique, cela permet de proposer un fonctionnement en boucle locale (mode restreint) ne reposant sur aucun processus de service.

Dans le sens ascendant, en revanche, on ne peut pas se permettre d'attendre qu'un processus utilisateur exécute un appel système pour traiter au niveau de la couche session les messages qui arrivent[3]. Ce traitement est donc systématiquement assuré par la partie basse du système MPCFS, dans la boucle de traitement du processus **`sockd`**.

La charge de travail qui incombe au processus **`sockd`** est donc relativement importante. En effet, elle concerne à la fois les traitements de la couche de protocole de niveau session et les traitements liés à la couche de protocole de niveau transport.

Au niveau de la couche transport, les traitements consistent (i) à réaliser les opérations de lecture et d'écriture sur le ou les *sockets* d'émission et de réception, (ii) à gérer le placement ou déplacement des messages sur les files de message de la connexion (décrites au paragraphe 6.6.1, page 150) et (iii) à gérer les échéanciers de la connexion (expiration des délais de retransmission, politique d'autorégulation, . . .).

Au niveau de la couche session, les traitements consistent (i) à prendre en charge l'émission et la réception des messages utilisateurs, (ii) à mettre en œuvre les protocoles de gestion répartie des groupes de communication (gestion des associations de groupes, calcul des intersections des arborescences des sous-groupes) et (iii) à gérer la synchronisation avec les processus locaux.

7.4.2.1 Description de l'algorithme de traitement d'une connexion active

L'algorithme résultant est assez complexe, car les traitements doivent d'une part, être ordonnés de façon à optimiser les performances du système sur le chemin critique de traitement des messages utilisateurs, et d'autre part, être réalisés en respectant les contraintes de la politique d'autorégulation.

3. Lorsque, par exemple, un message ne peut être accepté dans un sous-groupe par manque de place, il faut en aviser la machine expéditrice au plus vite afin qu'elle cesse ses émissions.

Cet algorithme, procède selon les sept étapes suivantes :

1. **Lecture de N_l messages** sur le *socket* de réception associé à la connexion. Les objectifs de cette première étape sont les suivants :

   – éliminer les messages malformés ;

   – libérer de la mémoire dans les tampons de réception du *socket* afin d'autoriser la réception de nouveaux messages[4] ;

   – libérer des emplacements dans la file d'émission des messages (par le traitement des accusés de réception) ;

   – mettre à jour l'estimation du temps d'aller-retour des messages (voir la description du protocole de transport, au paragraphe 8.2, page 196) ;

   Pour cela, le nombre de lectures pouvant être réalisées à chaque itération (N_l) est relativement élevé, mais reste limité. Il est élevé afin de permettre le traitement d'un grand nombre d'accusés de réception à chaque itération. Il est limité afin de satisfaire les contraintes de la politique d'auto-régulation et de permettre au système de résister au phénomène d'inondation. Les informations de contrôle de niveau transport contenues dans les en-têtes des messages (notamment les accusés de réception) sont traitées immédiatement. De cette façon, les messages courts, formés d'un unique en-tête de niveau transport, peuvent être éliminés dès cette étape. Les messages longs sont placés sur la file de remise en ordre, tant que leur numéro de séquence fait partie de la fenêtre de réception courante. Cette fenêtre permet d'assurer que le nombre total des messages qui sont en attente sur l'une des deux files de réception (file de remise en ordre et file d'attente de traitement) est borné. Les messages ne faisant pas partie de la fenêtre sont éliminés de façon silencieuse[5]. Dans le prototype actuel, la taille de la fenêtre de réception est fixée par défaut à 24 emplacements.

2. **Déplacement des éventuels messages en séquence** depuis la tête de la file de remise en ordre vers la queue de la file de traitement (un message est dit en séquence lorsque tous les messages qui le précèdent ont été reçu). Les messages sont placés sur la file de remise en ordre dans l'ordre croissant de leur numéros de séquence. Si des messages reçus sont en séquence (c'est-à-dire que tous les messages les précédant ont été reçus), ils apparaissent obligatoirement en tête de la file de remise en ordre.

3. **Affectation aux messages de contrôle prioritaires en attente d'émission des emplacements libérés sur la file d'émission** lors de la première étape (grâce au traitement des accusés de réception reçus). Il n'existe actuellement qu'un seul type de message prioritaire : il s'agit du message de niveau session indiquant que les émissions interrompues dans un sous-groupe (par manque de place dans le répertoire de stockage associé) peuvent reprendre. Ces messages de contrôle doivent être traités en priorité, afin de minimiser les délais de reprise des émissions sur les machines distantes.

4. Un *socket* dont le tampon de réception est plein rejete systématiquement tout nouveau message. Les messages rejetés sont définitivement perdus.

5. En fait, indirectement, ils peuvent déclencher l'envoi d'une notification de saturation demandant la diminution du débit en émission de l'autre machine.

4. **Traitement N_t messages**. Si la file des messages prêts à être traités contient des messages, les N_t premiers messages en attente sur cette file sont traités (au niveau session). Le traitement de chacun de ces messages peut exiger l'envoi d'un ou plusieurs messages de réponse. Par conséquent, pour qu'un tel message puisse être traité, il faut qu'il y ait suffisamment d'emplacements disponibles sur la file d'émission. Dans le cas contraire, lorsque la file d'émission des messages est pleine, le traitement des messages restants est reporté à l'itération suivante.

5. **Réveil des processus bloqués en émission.** S'il reste suffisemment d'emplacements sur la file d'émission pour accepter les messages de données des processus locaux, et que des processus sont bloqués en écriture dans l'attente d'un emplacement sur cette file, ils sont réveillés. Rappelons qu'afin de réduire le nombre des processus qui sont effectivement réveillés, ceux-ci sont répartis sur de multiples files d'attente (voir paragraphe 6.6.2, page 153).

6. **Émission de N_e messages**. Tant que le délai inter-émission (gigue) le permet, les N_e premiers messages en attente d'émission sont envoyés (si la gigue n'est pas nulle, au plus un message peut être envoyé). Si aucun message n'a été envoyé et que la couche transport requiert l'émission d'un message (un accusé de réception, une demande de retransmission) et que la gigue le permet, un message de contrôle de niveau transport est formé et envoyé.

7. **Calcul de la date de la prochaine itération**. Ce calcul tient compte d'une part, des messages qu'il reste à traiter et, d'autre part, des messages qu'il reste à émettre ou ré-émettre selon la date de dernière émission, la valeur de la gigue courante et du délai de retransmission courant.

Dans le prototype actuel, les valeurs des limites N_l, N_t et N_e sont initialisées par défaut à 32, 16 et 8 messages, respectivement. La valeur de N_e peut décroître, lorsque l'autre machine signale que ses files de réception sont pleines et que des messages ont du être rejetés. Lorsque N_e est à 1 et que cette diminution n'est toujours pas suffisante à réguler le débit d'arrivée des messages, la valeur de la gigue, initialisée à 0 par défaut, est augmentée progressivement. A l'inverse, la gigue diminue puis la valeur de N_e s'accroit progressivement, lorsque l'autre machine signale que ses files d'émission sont pleines et que les files de réception locales sont vides [6].

7.4.2.2 Analyse de l'algorithme de traitement d'une connexion active

Cette façon d'ordonner les traitements permet à l'algorithme d'être efficace dans les conditions idéales, tout en restant satisfaisantes lorsque les conditions deviennent difficiles.

Les conditions sont idéales lorsque la gigue est nulle et que l'occupation des files de messages est faible (inférieure aux valeurs limites N_l, N_t et N_e). Dans ce cas, la réception

6. Ce mécanisme de régulation, qui est surtout prévu pour permettre au système de s'adapter aux réseaux à très haut débit, n'a toutefois pas encore pu être testé et validé.

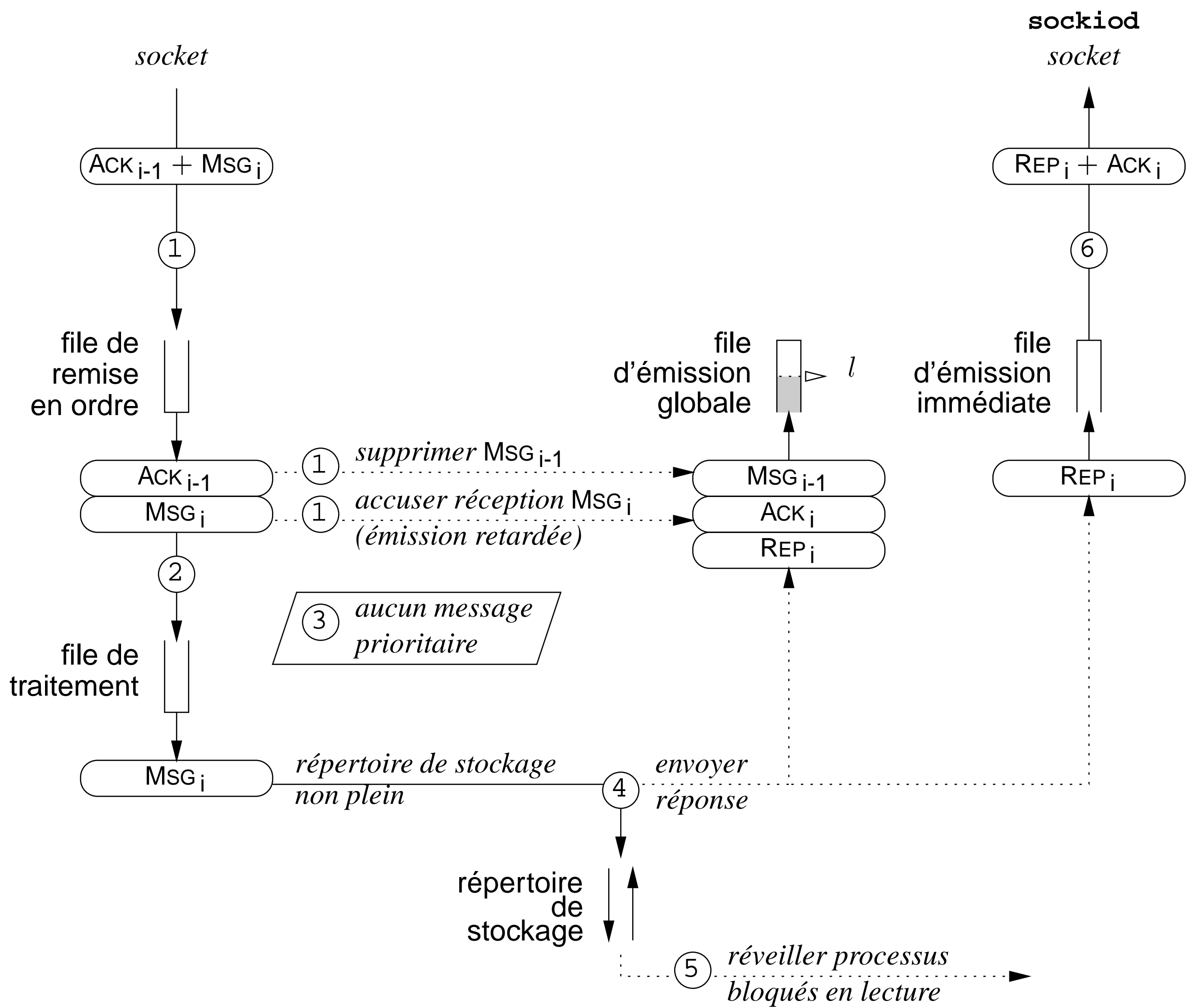


FIG. 7.4: *Réception et traitement d'un message dans le cas idéal.*

d'un message utilisateur, sa mise à la disposition des processus locaux, son acquitement aux niveaux session (acceptation ou refus dans le répertoire de stockage destination) et transport peuvent être réalisés au cours de la même itération (figure 7.4).

Etudions maintenant les situtions critiques et la façon dont elles sont contrôlées :

- **Inondation** : lorsqu'un message produit par un processus local est diffusé, de nombreuses réponses peuvent être reçues dans un laps de temps très court. Ces messages étant envoyés par des machines différentes, ils arrivent sur des connexions différentes. Le temps limité consacré au traitement de chaque connexion et la priorité accordée aux lectures d'en-tête (étape 1), permettent de minimiser les effets de ce phénomène d'innondation.
- **Émission continue** : lorsque des processus utilisateurs envoient des messages de façon continue, la file d'émission globale se remplit. Lorsque la file est pleine, les processus locaux sont endormis lors de l'opération d'écriture (mode d'émission semi-synchrone

évoqué au paragraphe 5.2.2.3, page 109), jusqu'à ce qu'un emplacement se libère. Comme nous l'avons vu au paragraphe 6.6.2 (page 153), l'utilisation de multiples files d'attente permet de minimiser le nombre de processus qui sont réveillés à chaque libération d'un emplacement.

- **Remplissage de la file d'émission globale** : comme nous venons de l'indiquer, lorsque des processus locaux émettent des messages de façon continue et/ou lorsque des messages sont reçus à une cadence élevée, la file d'émission globale se remplit. Lorsque cette file est pleine, les processus locaux sont bloqués lors de l'opération d'écriture. En revanche, il n'est pas souhaitable de bloquer les réceptions de façon prolongée, car cela aurait de fortes chances d'aboutir une situation d'interbloquage.

  En effet, quand la file d'émission est pleine, l'écoulement de la file de traitement se bloque dès qu'un message de cette file exigeant l'envoi d'un message de réponse arrive en tête de la liste pour être traité. La file de traitement se remplit donc à son tour. Lorsqu'elle est pleine, les messages de l'autre machine sont systématiquement rejetés, car ils ne font pas partie de la fenètre de réception. La file d'émission, puis la file de traitement de l'autre machine se remplissent à leur tour, ce qui aboutit à la situation d'interbloquage. Pour résoudre ce problème, nous avons fixé un ratio d'occupation de la file d'émission globale par les messages de données, les demandes d'association et les annonces de création de sous-groupes, en provenance des utilisateurs locaux (la valeur l sur la figure 7.4). Ces trois types de messages sont en effet les seuls types de messages dont la réception peut entraîner l'envoi d'un message de réponse. Dans le prototype actuel, ce ratio a été fixé à 0.5, ce qui signifie qu'au plus la moitié de la file peut être occupée par ces trois types de messages.

7.4.3 Associations de machines

L'association des groupes de communication de deux machines se fait à l'initiative de l'une des deux machines, lorsqu'un processus de l'une écrit le nom de l'autre machine dans le fichier d'association (`activate`) présent à la racine de l'arborescence du groupe (vu au paragraphe 5.3.2.10, page 125).

L'algorithme d'association est donc exécuté dans le cadre de l'appel système `sys_write()` invoqué sur le fichier `activate` du groupe. Cet algorithme s'exécute donc dans le contexte d'exécution noyau du processus utilisateur qui effectue la demande.

Il commence par former une requète pour le processus **`connd`**. Les objectifs de cette requète sont les suivants :

- vérifier la validité de la demande par rapport à la configuration établie par l'administrateur du système ;

- obtenir les éventuels paramètres de l'association ;

- obtenir la conversion en adresse IP du nom de la machine reçu en paramètre en.

L'algorithme vérifie ensuite que cette machine n'est pas déjà associée dans ce groupe. Si ce n'est pas la cas, l'algorithme détermine si une connexion est déjà active ou en train d'être établie entre les deux machines. Si aucune connexion n'est encore établie, l'algorithme d'association déclenche l'algorithme d'établissement d'une connexion (décrit au paragraphe 7.4.5). Lorsque la connexion est établie, l'algorithme déclenche le protocole d'association.

Ce protocole, que nous décrivons plus en détail au paragraphe 8.3.3 (page 210), procède en deux étapes :

1. Les machines échangent leurs identificateurs locaux pour le groupe de communication ; cet échange permet en particulier de vérifier que le groupe existe bien de part et d'autre ;
2. Les machines échangent leur arborescence de sous-groupes respectives. Pour cela, le protocole qu'elles utilisent implémente l'algorithme de calcul d'intersection d'arborescences réparties qui est présenté au paragraphe 8.3.5.

La transmission des messages émis par les processus de l'une des machines vers l'autre machine débute dès la fin de la première étape, pour les messages transmis au niveau de la racine de l'aborescence des sous-groupes du groupe. Pour les autres niveaux de l'arborescence, les échanges ne se mettent en place qu'au fur et à mesure que l'algorithme de la deuxième étape progresse.

7.4.4 Dialogue entre les processus utilisateurs et le processus `connd`

Nous venons de voir que lors du traitement d'une demande association, une requète était adressée au processus de service **`connd`**. Nous allons maintenant décrire plus précisément le mécanisme utilisé pour la transmission de cette requète. Le dialogue entre le processus utilisateur exécutant l'algorithme d'association ou de connexion et le processus de service **`connd`** s'opère directement dans le contexte du noyau. Ce dialogue s'effectue au travers (i) d'une structure décrivant la requête (appelée **`requête`**), (ii) d'une variable d'état décrivant l'état d'avancement du traitement de la requête et (iii) de files d'attente permettant de synchroniser les processus. Il n'existe, à tout moment, qu'une seule instance active de la structure **`requête`** dans le système. Cette instance est accessible par tous les processus dans le contexte d'exécution du noyau. Elle est utilisée pour les deux types de requêtes reconnues par le processus **`connd`** : vérification d'une association et établissement d'une connexion.

Cette requête est formée des champs suivants :

- **`type`** : LOCALE, DISTANTE ou ASSOCIATION ;
- **`id_local`** : identifiant local de la connexion[7] ;
- **`autre_machine`** : structure décrivant l'autre machine (identifiant, nom, adresse IP, etc) [7].

7. Voir la description de la structure associée à une connexion donnée au paragraphe 6.6, page 149.

– **résultat** : résultat du traitement de la requête (paramètres de la connexion en cas de succès, code d'erreur en cas d'échec) ;

Les trois files d'attente utilisées pour synchroniser les processus sont les suivantes :

– **connd_attente**, qui est utilisée par le processus **connd** lorsqu'il est disponible pour traiter une requête. Le processus **connd** se met en attente sur cette file à partir de son contexte utilisateur, en invoquant l'appel système `sys_select()` sur l'un des deux fichiers spéciaux de l'arborescence MPCFS dédiés au processus **connd** (l'un des deux fichiers est utilisé pour le traitement des requêtes locales, et l'autre pour le traitement des requêtes distantes) ;

– **attente_requête**, qui est utilisée par les processus qui attendent que le descripteur de requête soit disponible ;

– **attente_terminaison**, qui est utilisée par les processus qui attendent le résultat d'une requête en cours de traitement ;

La principale variable d'état employée, est la variable **status_requête** qui indique l'état d'avancement de la requête :

– **DISPONIBLE**, lorsque la requête est disponible ;

– **INITIALISÉE**, lorsque la requête est initialisée mais n'est pas encore prise en charge par le processus **connd** ;

– **EN_COURS**, lorsque la requête est en cours de traitement ;

– **TERMINÉ**, lorsque le traitement de la requête est terminé.

L'algorithme 7.7 (page 188) décrit la partie de l'algorithme de dialogue exécutée par un processus utilisateur désirant soumettre une requête au processus **connd**. Notons que cet algorithme permet à un processus désirant soumettre une requête identique à une autre requête déjà en cours de traitement, de se mettre en attente du résultat de la requête courante (plutôt que de soumettre à nouveau la même requête). Ceci permet de résoudre en partie les problèmes de concurrence qui se posent lors de l'établissement d'une connexion.

Du côté du processus **connd**, lorsque la présence d'une nouvelle requête locale est détectée, cette requête doit tout d'abord être transmise du contexte d'exécution du noyau vers le contexte d'exécution utilisateur. Le processus **connd** réalise ce transfert au moyen de l'appel système `sys_read()`, invoqué sur le fichier de l'aborescence MPCFS `/mpc/.connd_local`. Ce fichier est dédié au traitement des requêtes locales de conversion ou de connexion[8]. Le traitement réalisé au cours de cet appel système est très simple : (i) la variable **status_requête** est passée de l'état **INITIALISÉE** à l'état **EN_COURS** et (ii)

---

8. Le processus **connd** utilise un autre fichier de l'arborescence MPCFS, appelé `/mpc/.connd_remote` pour le traitement des requêtes distantes.

ALG. 7.7 Dialogue avec le processus de service **connd** (partie processus utilisateur)

```
fonction dialogue_vers_connd(nouvelle_requête)
début
   tant que (status_requête ≠ DISPONIBLE) faire
      si (requête_identique(nouvelle_requête,requête)) alors
         {Un autre processus a lancé la même requête}
         copie←attendre_résultat()
         retour copie
      sinon
         {Un autre processus a lancé une requête différente}
         dormir_sur_file(attente_requête)
      finsi
   fait

   requête←nouvelle_requête
   réveiller_file(connd_attente)
   copie←attendre_résultat()
   retour copie
fin

fonction attendre_résultat()
début
   tant que (status_requête ≠ TERMINÉ) faire
      dormir_sur_file(attente_terminaison)
   fait

   si (file_vide(attente_terminaison)) alors
      status_requête←DISPONIBLE
   finsi

   si (non file_vide(attente_requête)) alors
      réveiller_file(attente_requête)
   finsi

   retour requête
fin
```

la structure de la requête est copiée dans le contexte utilisateur du processus à l'adresse indiquée en paramètre de l'appel système.

Lorsque le traitement de la requête en contexte utilisateur est terminé, la copie de la requête mise à jour avec le résultat dans le contexte utilisateur est transmise en sens inverse, vers le contexte d'exécution du noyau. Le processus **connd** invoque, pour cela, l'appel système `sys_write()` sur le même fichier que précédemment (`/mpc/.connd_local`). Le traitement de cet appel système par le noyau est à peine plus compliqué que dans l'autre sens : (i) la requête est copiée à partir de l'adresse reçue en paramètre de l'appel système, (ii) la variable **status_requête** est passée de l'état **EN_COURS** à l'état **TERMINÉ** et (iii) les processus endormis sur la file **attente_terminaison** sont réveillés.

7.4.5 Établissement des connexions

L'établissement d'une connexion MPCFS entre deux machines est pris en charge par le processus **connd**. Sur chacune des machines, ce processus joue à la fois le rôle de client et de serveur. La machine qui prend l'initiative d'établir la connexion est le client et la machine sollicitée est le serveur. Le contact est établi entre les deux machines par un protocole de type RPC, construit à l'aide de l'outil `rpcgen` [Sun Microsystems, Inc. 90].

L'algorithme d'établissement d'une connexion est assez complexe. En premier lieu parce qu'il est asymétrique : l'algorithme de la partie cliente est différent de celui de la partie serveur. La partie cliente est déclenchée par l'algorithme d'association des groupes de communication (que nous venons de décrire au paragraphe 7.4.3), alors que la partie serveur est déclenchée par la réception d'une requête RPC, dans le contexte d'exécution utilisateur du processus **connd**. En second lieu, parce qu'il s'exécute dans de multiples contextes : d'une part, dans le contexte noyau du processus utilisateur ayant déclenché le mécanisme d'association et d'autre part dans les contextes noyau et utilisateur du processus de service **connd** (traitement de la requête RPC).

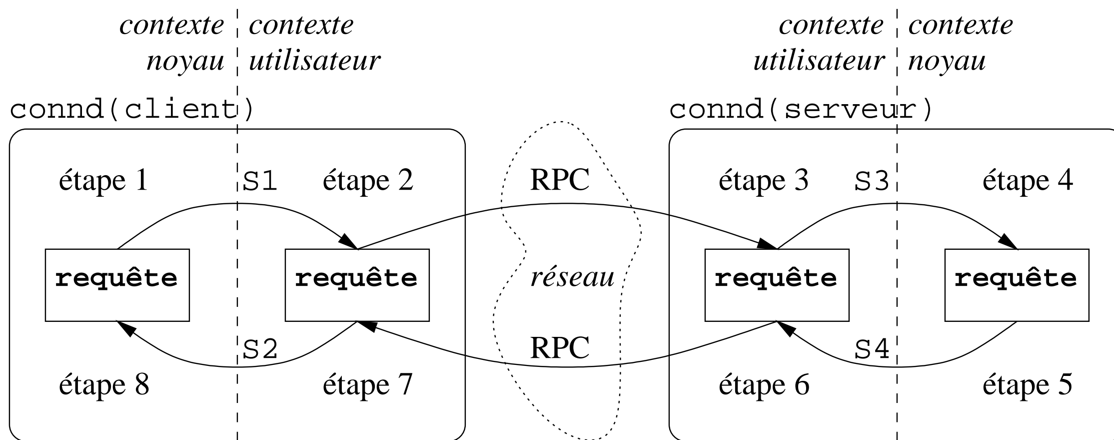
Avant de décrire de quelle façon ces problèmes sont traîtés, au paragraphe 7.4.5.2, nous décrivons d'abord le fonctionnement général de l'algorithme, dans le paragraphe suivant.

7.4.5.1 Description de l'algorithme d'établissement des connexions

L'objectif de cet algorithme est de faire en sorte que les deux machines s'échangent les paramètres de la connexion, afin d'initialiser les champs de leurs structures respectives de connexion (décrite au paragraphe 6.6, page 149).

Ces paramètres sont échangés au moyen de la structure de requête décrite au paragraphe 7.4.4. L'algorithme de connexion procède en huit étapes, au cours desquelles la requête effectue un aller-retour entre le contexte d'exécution du noyau de la machine cliente et le contexte d'exécution noyau de la machine serveur (figure 7.5) :

1. Le processus **connd** de la machine cliente détecte la requête de connexion. Il initialise une structure de connexion, met à jour la requête avec les paramètres de cette connexion (identifiant de la connexion, adresse IP et nom de l'autre machine) et prépare la transmission de la requête vers son contexte d'exécution utilisateur en indiquant que des données sont lisibles sur le fichier `/mpc/.connd_local`.




```
S1 : sys_read(/mpc/.connd_local,requête)
S2 : sys_write(/mpc/.connd_local,requête)
S3 : sys_write(/mpc/.connd_remote,requête)
S4 : sys_read(/mpc/.connd_remote,requête)
```

FIG. 7.5: *Établissement d'une connexion MPCFS*

2. En contexte utilisateur, le processus **connd** lit la requête dont il a détecté la présence sur le descripteur du fichier `/mpc/.connd_local`. Il vérifie alors que la connexion entre les deux machines est bien autorisée par le fichier de configuration `/etc/mpc_exports`. Si ce n'est pas le cas, la connexion échoue et l'algorithme passe directement à l'étape 7. Dans le cas contraire (succès), un *socket* est initialisé avec un nouveau numéro de port pour la connexion. Ce numéro de port est inséré dans la requête, qui est ensuite envoyée dans un message RPC au processus **connd** de la machine serveur.

3. En contexte utilisateur, le processus **connd** de la machine serveur reçoit le message RPC contenant la requête de la machine cliente. Il vérifie que de son côté, la connexion entre les deux machines est bien autorisée par son fichier `/etc/mpc_exports`. En cas d'échec de la vérification, il passe directement à l'étape 6. Dans le cas contraire (succès), il crée et initialise à son tour un *socket* avec un nouveau numéro de port. Puis il transmet la requête de son contexte d'exécution utilisateur vers son contexte d'exécution noyau par une écriture sur le fichier `/mpc/.connd_remote`. Sinon, il passe directement à l'étape 6.

4. L'écriture sur le fichier `/mpc/.connd_remote` a fait basculer le processus en contexte d'éxécution noyau. Après avoir copié la requête dans le contexte du noyau, le processus initialise une structure de connexion et met à jour les champs concernant cette connexion dans la requête. Puis, il indique que ces données sont lisibles sur le fichier `/mpc/.connd_remote`. La connexion est alors opérationnelle pour la machine serveur.

5. Ayant détecté la présence de données à lire sur le descripteur du fichier

`/mpc/.connd_remote`, le processus **connd** copie la requête vers son contexte utilisateur.

6. Le processus **connd** de la machine serveur retourne la requête qu'il vient de traiter à la machine cliente, dans un message de réponse RPC.

7. Le processus **connd** de la machine cliente reçoit la réponse à son message RPC, contenant la requête mise à jour par la machine serveur. Il écrit cette requête dans le fichier `/mpc/.connd_local`.

8. L'écriture sur le fichier `/mpc/.connd_local` a fait basculer le processus en contexte d'exécution noyau. Il copie la requête depuis le contexte utilisateur et met à jour la structure de connexion initialisée à l'étape 1 avec les informations figurant dans la requête reçue. La connexion est alors opérationnelle sur la machine cliente.

7.4.5.2 Gestion de la concurrence

Les multiples changements de contexte opérés durant le déroulement de cet algorithme se traduisent par de nombreux problèmes de concurrence entre processus. À chaque changement de contexte, de nouveaux processus utilisateurs peuvent en effet déclencher à leur tour le mécanisme d'association et, par suite, requérir l'établissement d'une connexion. Les conflits résultants ont d'autant plus de chances de se produire que le temps global requis pour l'exécution de l'algoritme d'établissement d'une connexion peut être long (au minimum, le temps d'aller-retour d'un message entre les deux machines).

La résolution des conflits entre machines est en partie laissée à la charge du mécanisme de RPC, qui fonctionne de façon synchrone. Lorsqu'une machine envoie une requête de connexion à un serveur, elle passe en mode client et sa fonction de serveur est momentanément désactivée. Les requêtes de connexion que reçoit cette machine pendant ce laps de temps sont ignorées. Lorsqu'une requête est ignorée trop longtemps, elle finit par échouer. Dans la plupart des cas, ce traitement simple est suffisant, car les requêtes momentanément ignorées finissent par aboutir. Mais dans certaines circonstances ce traitement ne suffit pas. Par exemple, il peut arriver que deux machines qui s'envoient simultanément une requête de connexion, s'interdisent mutuellement le traitement de leur requête respective. L'algorithme de connexion doit donc prévoir une solution pour résoudre ce conflit (ce qui n'est pas le cas dans le prototype actuel). Une solution simple consisterait, par exemple, à réessayer plusieurs fois d'établir la connexion, en introduisant un délai aléatoire entre chaque essai afin de désynchroniser les deux machines.

La résolution des conflits entre processus d'une même machine est assurée par le protocole de dialogue entre les processus utilisateurs et le processus **connd**. Comme nous l'avons expliqué au paragraphe 7.4.4, ce protocole assure l'exclusion mutuelle des processus en n'autorisant la soumission que d'une seule requête à la fois. Un processus désirant soumettre une requête alors qu'une requête précédente est en cours de traitement doit donc attendre la complétion de la requête courante.

7.5 Conclusion

Dans ce chapitre, nous avons décrit les principaux algorithmes que nous avons conçu et mis en œuvre dans le prototype actuel de MPCFS pour le système d'exploitation Linux. Nous avons d'abord décrit les algorithmes de la partie haute du système, déclenchés lors des interactions des utilisateurs avec le système de fichiers virtuel. Ces algorithmes, qui s'appuient exclusivement sur le mécanisme de VFS, sont relativement simples. Leur seule mise en œuvre permet de faire fonctionner le système en boucle locale, c'est-à-dire entre processus d'une même machine.

Puis, nous avons décrit la partie basse, chargée de la gestion des messages et des protocoles de communication entre machines. A ce niveau, en revanche, les algorithmes sont beaucoup plus complexes, car (i) les protocoles (que nous décrivons plus en détail au chapitre suivant) sont sujets aux interbloquages et (ii) les fonctionnalités requises pour les mettre en œuvre ne sont pas toutes accessibles depuis le contexte d'exécution du noyau.

Comme nous l'avons vu, ces algorithmes sont, dans l'ensemble, très généraux. Ils peuvent être adaptés à la majorité des systèmes UNIX existants, dès lors que ces derniers supportent un mécanisme de type VFS.

En effet, les seuls services et fonctions du noyau auxquels nous avons recours, en plus de la surcharge des appels systèmes portant sur les fichiers de l'arborescence, sont les suivants :

– L'accès aux fonctions d'émission et de réception sur les *sockets* ;

– Les files d'attente. Ces mécanismes sont largement disponibles sur tous les systèmes UNIX, souvent même sous des formes plus élaborées que celle que nous avons utilisé. Rappelons, par exemple, que Solaris permet de ne réveiller que le processus de plus haute priorité d'une file d'attente.

Toutefois, l'adaptation du système MPCFS à un système ne permettant pas l'accès à ces fonctions (ou à des fonctions similaires) depuis le contexte du noyau est toujours possible. Il suffit pour cela de déporter tout ou partie des traitements en contexte utilisateur et de mettre en place un protocole de dialogue entre le processus service réalisant ces fonctions et le noyau. Les gains en performance que l'on peut envisager avec une solution s'exécutant en contexte utilisateur sont naturellement moins importants. Mais le système n'en reste pas moins intéressant, car ses autres avantages, tels que la transparence ou la simplicité d'utilisation, par exemple, ne sont pas remis en cause.

D'une façon générale, les problèmes techniques liés à la mise en œuvre du système MPCFS sont de même nature que ceux qui se posent pour la réalisation du système de fichiers NFS. En d'autre termes, tout système d'exploitation qui est actuellement capable de supporter le système de fichiers NFS, est théoriquement capable de supporter le système MPCFS. La portabilité du système MPCFS s'étend largement au-delà du monde UNIX, et peut être envisagée sur des système tels que Windows NT (aussi très présent sur les grappes de stations de travail), voire même MS-DOS.

Chapitre 8

Protocoles

8.1 Introduction

Dans ce chapitre, nous présentons les deux couches de protocole que nous avons développées dans le prototype MPCFS. Ces couches s'appuient sur la suite de protocoles TCP/IP, et sur le protocole UDP/IP en particulier, auquel nous accédons depuis le noyau, au travers de l'API des *sockets*.

Les fonctionnalités offertes par cette couche de transport étant minimales (point-à-point non-fiable et non ordonné, de type commutation de paquets), nous avons du compléter ce niveau transport par l'ajout d'une sur-couche assurant la transmission multipoints fiable et ordonnée des messages (FIFO selon la source). Ce type de service, très complexe à optimiser de part la diversité des besoins auxquels il est susceptible de répondre, n'est pas encore disponible dans la suite de protocole TCP/IP. Le protocole que nous présentons n'a donc pas la prétention de fournir une solution optimale à ce problème, mais répond simplement à la nécessité de disposer d'un tel service pour permettre le fonctionnement de l'ensemble du prototype.

Au niveau supérieur, la transmission des messages reçus des utilisateurs s'appuie donc sur cette sur-couche de niveau transport. Mais à coté de ces messages utilisateurs, notre système utilise aussi un certain nombre de messages de contrôle, de façon interne, afin notamment d'assurer la gestion répartie des groupes de communication. La diversité de ces messages nous a donc amené à construire une deuxième sur-couche de protocole, au-dessus de la première. Reprenant la classification OSI-7, cette deuxième couche de protocole, relativement indépendante de la première, peut être assimilée à une couche de niveau session.

La figure 8.1 illustre cette organisation multi-couches du système MPCFS. Au niveau le plus haut de ce diagramme, on retrouve les trois types d'interactions des utilisateurs avec le système MPCFS :

- Les **demandes d'association** de groupes de communication sont déclenchées à la suite d'une écriture sur le fichier `activate` d'un groupe. Au niveau inférieur, une demande d'association peut déclencher à son tour le **protocole d'établissement d'une connexion** entre deux machines. Elle entraîne ensuite l'envoi d'une demande d'association. Cette demande permet de vérifier que le groupe existe bien sur les deux machines, et, dans l'affirmative, permettre l'échange de leurs identifiants respectifs du groupe. Si cette demande est acceptée, les deux machines font alors appel au **protocole d'échange des identifiants de sous-groupes** afin de calculer l'intersection de leur deux arborescences de sous-groupes.

- Les **créations de sous-groupes** sont obtenues à la suite de la création d'un répertoire de multiplexage dans l'arborescence de l'un des processus membres d'un groupe. Dans un premier temps, cette opération entraîne la **diffusion du nom et de l'identifiant du sous-groupe** auprès de toutes les machines ayant défini le sous-groupe parent. Puis, le **protocole d'échange des identifiants de sous-groupes** est déclenché avec chacune des machines sur lequel le sous-groupe qui vient d'être créé existe, afin de mettre à jour l'intersection des arborescences de sous-groupes des deux machines. Ce protocole

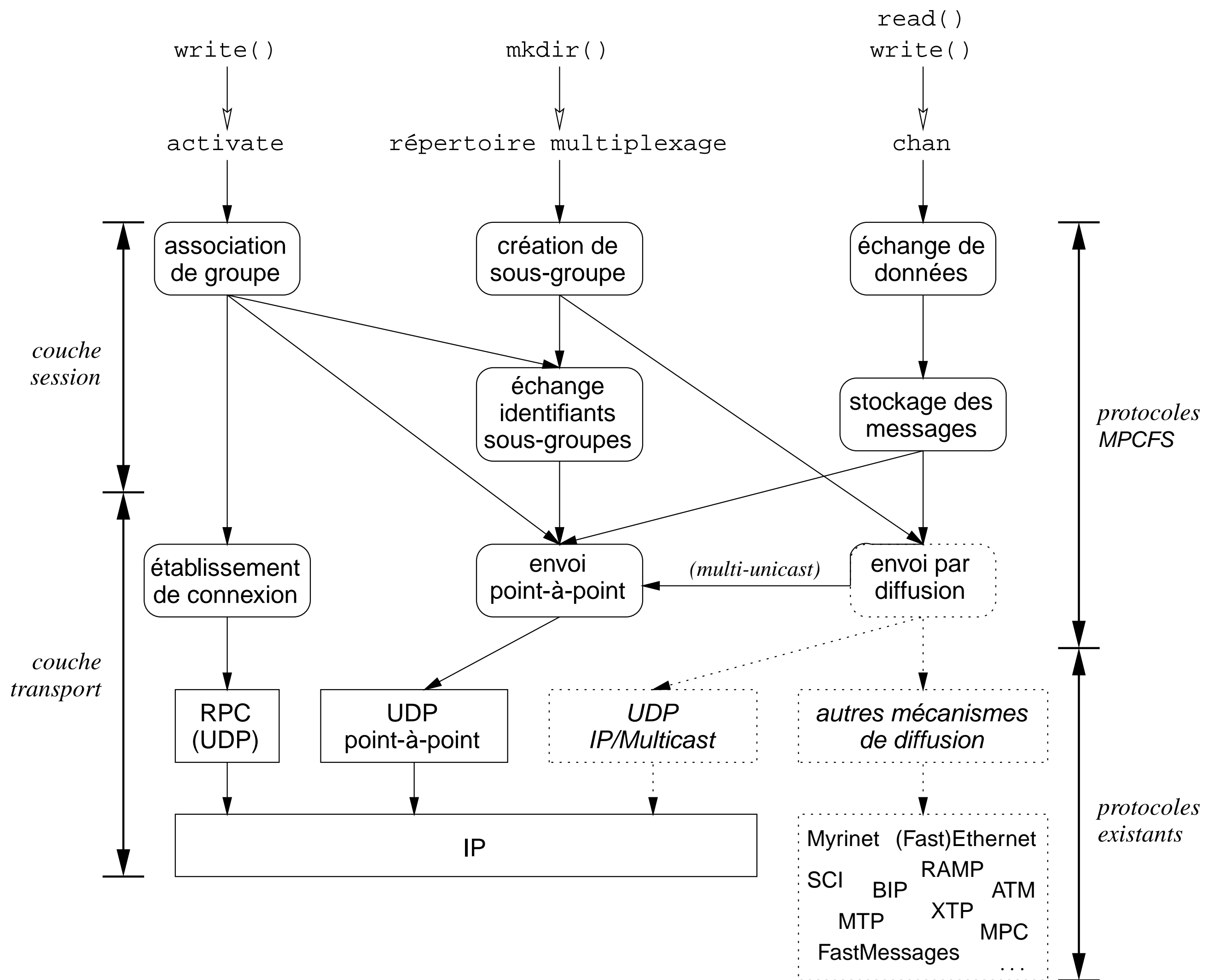


FIG. 8.1: *Organisation des couches de protocoles dans MPCFS*

est déclenché en partant du nouveau sous-groupe, afin d'assurer que des parties de l'arborescence ne soient pas oubliées, lorsque, par exemple, plusieurs sous-groupes imbriqués sont créés en un laps de temps très court).

- Les **échanges de données** sont obtenus à la suite d'une opération de lecture ou d'écriture sur l'un des fichiers `chan` de l'arborescence MPCFS. En réception, les données sont lues à partir des répertoires de stockage. En émission, les données sont placées dans les répertoires de stockage pour être diffusées auprès de chaque machine faisant partie du sous-groupe utilisé pour l'envoi. Certaines machines destinataires peuvent toutefois refuser les messages de données qui leur sont envoyés, par manque de place dans leur propre répertoire de stockage pour ce sous-groupe. Ce refus implique la mise en place d'un **protocole de signalisation et de retransmission au niveau des répertoires de stockage**.

Au centre de ce diagramme, se trouve le protocole de communication point-à-point fiable et ordonné de MPCFS. Ce protocole fonctionne de façon très similaire au protocole TCP, mais au-dessus du protocole UDP. Ce protocole permet l'échange des messages point-à-point émanant des protocoles des niveaux supérieurs. Ces messages point-à-point sont (i) les requêtes d'association, (ii) les messages du protocole d'échange des identifiants de sous-groupes, (iii) les retransmission des messages de données et (iv) les messages de signalisation des répertoires de stockage (message d'acceptation ou de refus, et messages de reprise des émissions). Lorsque le mécanisme de diffusion est le *multi-unicast* (ce qui est le cas dans le prototype actuel), ce protocole se charge aussi de l'envoi des messages de données et des annonces de création de sous-groupes vers chacune des machines destinataires. Les boîtes en tracé pointillé sur le diagramme représentent les mécanismes qui ne sont pas encore implantés dans le prototype.

8.2 Couche transport

L'objectif de la couche de protocole de niveau transport est d'assurer la transmission fiable et ordonnée des messages depuis une machine vers une ou plusieurs autres machines.

Les communications du système MPCFS s'appuient sur deux mécanismes de communications élémentaires : un mécanisme de diffusion et un mécanisme de transmission point-à-point. Selon les capacités du réseau, le mécanisme de diffusion peut être de type *broadcast*, *multi-cast* ou *multi-unicast*. Dans ce derniers cas, le système n'a besoin que du mécanisme de communication point-à-point.

8.2.1 Choix du mécanisme de communication point-à-point

En ce qui concerne le mécanisme de communication point-à-point, nous avons choisi de construire notre protocole au-dessus du protocole UDP/IP. Ce protocole offre en effet un compromis intéressant sur le plan des fonctionnalités offertes : il nous permet d'assurer un fonctionnement indépendant du ou des réseaux locaux sous-jacents. Il permet la transmission de messages raisonnablement longs car il bénéficie de la fragmentation et de la reconstruction

des messages réalisée au niveau IP. Par défaut, il prend en charge le calcul des mots de parité, ce qui assure l'intégrité des transmises ; mais cette fonctionnalité peut être désactivée en cas de nécessité (*multi-unicast*). Enfin, ce protocole permet d'exploiter directement les fonctionnalités du protocole IP/Multicast, pour mettre en place un mécanisme de diffusion.

Les autres possibilités que nous avons envisagées pour mettre en œuvre le mécanisme de communication point-à-point du système MPCFS sont de construire le protocole (i) directement au dessus de la couche liaison, (ii) au dessus de la couche réseau et (iii) au dessus du protocole TCP/IP. Mais comme nous l'expliquons dans les paragraphes qui suivent, ces solutions sont en général moins intéressantes que celle que nous avons choisie.

Avantages et inconvénients de la couche de niveau liaison. Cette solution permet certes d'exploiter au mieux les performances du réseau sous-jacent, en minimisant le nombre des étapes nécessaires au traitement d'un message. Dans certains cas, elle permet aussi de tirer partie des capacités spécifiques d'un réseau local, comme la capacité de diffusion (*broadcast*) ou de multi-distribution (*multicast*) des réseaux Ethernet.

Mais en contrepartie, elle entraîne une dépendance vis-à-vis d'un type de réseau local, et elle limite l'extensibilité du système, puisque par définition, la couche de liaison ne sait opérer qu'à l'échelle d'un unique réseau local. Par ailleurs, elle rend la conception du protocole beaucoup plus difficile, car elle nous oblige à prendre en charge des opérations délicates telles que la fragmentation et la reconstruction des messages, lorsque la taille dépasse le MTU (*Maximum Transfert Unit*) du réseau.

Cette solution n'est toutefois pas définitivement rejetée, car ces inconvénients ne sont pas toujours rédhibitoires. Tout d'abord, la majorité des réseaux locaux se conforment au standard IEEE 802, ce qui minimise la dépendance évoquée. Ensuite, dans le cas des grappes de stations de travail, un fonctionnement inter-réseaux n'est pas toujours indispensable.

Cette solution peut donc s'envisager dans un deuxième temps, lors d'une phase d'optimisation du système MPCFS, notamment afin d'exploiter au mieux les performances d'un réseau à haut débit tel que Myrinet, par exemple.

Avantages et inconvénients de la couche de niveau réseau (c'est-à-dire au même niveau que UDP ou TCP). Cette solution est intéressante sur le plan des performances, car elle permet de minimiser le chemin critique de traitement des messages tout en disposant de fonctionnalités minimales, telles que la fragmentation et la reconstruction des messages. Ces fonctionnalités restent toutefois limitées, en particulier en ce qui concerne le multiplexage des messages ou l'accès au protocole IP/Multicast. De plus, la mise en œuvre d'un nouveau protocole à ce niveau est délicate. Elle exige soit de modifier le noyau, soit d'ouvrir un socket en mode `RAW` afin de recevoir tous les messages du protocole IP. La première solution est difficile à réaliser, et ne permet plus au prototype d'être installé sans une recompilation du noyau. La seconde se traduit par un sur-coût conséquent sur le chemin critique de traitement de l'ensemble des datagrammes IP reçus par le système. Enfin, l'utilisation directe d'IP nous oblige à prendre systématiquement en charge le calcul des mots de contrôle, afin d'assurer l'intégrité des données transmises.

Avantages et inconvénients du protocole TCP/IP. Le mode de transmission fiable et ordonné du protocole TCP est certes un argument de poids en sa faveur. Mais parmi les nombreuses fonctionnalités offertes par ce protocole, qui sont en général fort utile, certaines se révèlent dans notre cas pénalisantes :

- *TCP est orienté connexion* : cette propriété est intéressante pour MPCFS. Toutefois, les connexions TCP sont exclusivement point-à-point, ce qui interdit l'accès à des protocoles multipoints tel IP/Multicast. En revanche, IP/Multicast est prévu pour fonctionner au travers du protocole UDP.
- *TCP ne préserve pas les frontières des message* : cette propriété est indispensable pour un système tel que MPCFS. La préservation des frontières de message permet de garantir à l'application qui émet un message par une unique écriture, que ce message sera reçu, de façon atomique, au travers d'une unique lecture par ses destinataires. De cette façon, l'entrelacement des données reçues de plusieurs machines peut être facilement contrôlé au niveau applicatif. L'utilisation de TCP nous obligerait donc à insérer des informations de contrôle dans le flot de données des applications, et à mettre en place un mécanisme de reconstruction des messages à leur arrivée. L'ajout d'une telle sur-couche de fragmentation et de reconstruction, au-dessus du protocole TCP, ne pose aucun problème technique en elle-même. En revanche, elle se traduirait inévitablement par une augmentation du délai de mise à disposition des messages.
- *TCP assure de façon automatique et systématique le contrôle de l'intégrité des données reçues*, à l'aide de mots de parité (*checksums*) : au moment de l'émission, TCP calcule le mot de parité sur les données qu'il va transmettre et l'associe au message. En réception, le destinataire effectue à son tour le calcul sur les données reçues et compare son résultat avec le mot de contrôle associé au message par l'expéditeur. Si une différence est détectée, c'est que les données ont été altérées lors de la transmission. Ce mécanisme est donc indispensable pour assurer la fiabilité des transmissions. En revanche le calcul des mots de parité est un calcul coûteux, qui doit être évité lorsqu'il peut l'être. Or, c'est justement le cas lorsqu'un même message doit être émis successivement vers plusieurs destinataires (*multi-unicast*) : dans ce cas, le calcul n'a besoin d'être fait qu'une seule fois. A l'inverse d'UDP, TCP ne permet pas cette optimisation, car sa fonction de calcul des mots de parité ne peut pas être désactivée.

8.2.2 Choix du mécanisme de communication multipoints

Le prototype ne supporte pour l'instant qu'un seul mécanisme de communication multipoints : le *multi-unicast*. Toutefois, le choix du protocole UDP pour la mise en place de notre protocole de communication point-à-point fiable et ordonné, nous permet aisément d'envisager l'utilisation du protocole IP/Multicast.

En effet, l'émission et la réception de messages IP/Multicast sont réalisées au travers de *sockets* de type UDP, c'est-à-dire initialisés pour la famille de protocole IP (`AF_INET`) et le type datagramme (`SOCK_DGRAM`). L'association d'un *socket* UDP avec un groupe IP/Multicast est ensuite obtenue par une simple reconfiguration du *socket* (à l'aide de la fonction `setsockopt()`).

D'autres solutions, a priori plus efficaces que le *multi-unicast*, sont bien-sûr envisageables pour construire notre mécanisme de communication multipoints. On peut notamment considérer les mécanismes de diffusion et de multi-distribution au niveau de la couche de liaison, ou bien sûr, le protocole IP/Multicast. Mais outre le fait que ces solutions ne soient pas toujours applicables, elles ont aussi leurs inconvénients.

Diffusion au niveau de la couche de liaison. Sur les réseaux de type bus ou anneau à jeton (Ethernet ou Token Ring, par exemple), cette solution permet de minimiser le nombre de messages émis, puisque chaque message n'a besoin d'être transmis qu'une seule fois. En revanche, sur les réseaux point-à-point (Myrinet, par exemple), cette solution n'apporte pas d'amélioration par rapport au *multi-unicast*.

Par ailleurs, cette solution comporte de nombreux inconvénients. Tout d'abord, elle oblige chacune des machines du réseau local à traiter, au niveau logiciel, chacun des messages diffusés. Lorsque le volume des messages diffusés est important, cette surcharge de travail est donc particulièrement pénalisante pour les machines qui ne participent pas aux communications multipoints du système MPCFS. Ensuite, si la diffusion au niveau liaison permet de diminuer le nombre des messages transmis, elle n'implique pas nécessairement une utilisation efficace du réseau. En effet, les réseaux locaux tel que Ethernet, lorsqu'ils sont de taille importante, sont souvent subdivisés en sous-réseaux (dans le cas d'Ethernet, on les appelle des domaines de collision), grâce à des ponts filtrants. Ces ponts permettent de cloisonner le traffic point-à-point, en limitant la propagation des messages aux seuls sous-réseaux qu'ils doivent emprunter pour atteindre leur destinataire. Dans le cas d'une diffusion, puisque toutes les machines du réseau sont destinataires du message, celui-ci doit être transmis sur tous les sous-réseaux. L'utilisation intensive de la diffusion conduit donc rapidement à un phénomène d'inondation, particulièrement dramatique lorsque la bande passante des sous-réseaux diffère (lorsque, par exemple, certains domaines de collisions fonctionnent à 10 MBits/s et d'autres à 100 MBits/s).

Multi-distribution (*multicast*) au niveau de la couche liaison. La norme IEEE 802 réserve l'ensemble des adresses dont l'octet de poids fort est égal à 1 (01-xx-xx-xx-xx-xx) pour des trames qui s'adressent à un groupe de destinataires (nous les appelons des "trames *multicast*"). Sur un réseau Ethernet, de la même façon que les trames de diffusion, les trames *multicast* sont envoyées sur l'ensemble des domaines de collision. Par conséquent, elles n'apportent pas de solution au problème d'inondation. Néanmoins, à la différence des trames diffusées, les trames *multicast* peuvent être filtrées à destination, par le matériel (l'adaptateur réseau). En principe, elle n'entraînent donc pas de surcharge pour les machines qui n'en sont pas destinataires. Néanmoins, un filtrage parfait impose à l'adaptateur de savoir mémoriser un nombre arbitrairement grand d'adresses *multicast*. En pratique, comme la multi-distribution au niveau liaison n'est pas un mécanisme très usité (du moins jusqu'à l'apparition du protocole IP/Multicast, ce dont nous discutons au point suivant), la capacité de filtrage des adaptateurs Ethernet est souvent limitée. Dans le pire des cas, ils ne savent qu'accepter ou refuser l'ensemble des trames multipoints ; dans le meilleur des cas, ils disposent d'une mémoire leur permettant de stocker des adresses de groupes et/ou utilisent une fonction de

hachage sur les adresses de de *multicast* utilisées, pour autoriser ou interdire en bloc des sous-ensembles de l'espace d'adressage *multicast*[1].

La multi-distribution au niveau liaison, quand elle est disponible, n'est donc pas toujours un mécanisme satisfaisant. Dans certains cas, son utilisation peut toutefois s'avérer intéressante. C'est par exemple le cas, lorsque le système MPCFS est utilisé comme support de communication dans une grappe de stations de travail, dont le réseau d'exploitation est indépendant ou délimité par des routeurs de niveau 3 (routage au niveau de la couche réseau).

Mais, de la même façon que pour les communications point-à-point, l'accès aux fonctionnalités du réseau au niveau de la couche liaison est difficile, car il nous oblige à prendre en charge un certain nombre de traitements comme la fragmentation et la reconstruction des messages. Or, comme nous allons le par la suite, le protocole IP/Multicast [Deering 89] permet d'accéder aux fonctionnalités de multi-distribution proposées au niveau des couches de liaison, tout en assurant un service de niveau transport, puisque ce protocole s'appuie sur UDP.

Multi-distribution au niveau de la couche réseau (IP/Multicast). Comme nous l'avons au paragraphe 3.3.1 (page 46), IP/Multicast est un protocole de multi-distribution offrant le niveau minimal de qualité de service : il est non connecté, non fiable et non ordonné. À l'échelle d'un réseau local, ce protocole réutilise la capacité de multi-distribution des réseaux sous-jcents lorsqu'elle est disponible.

IP/Multicast souffre donc, en principe, des mêmes inconvénients que la multi-distribution au niveau de la couche liaison. Toutefois, les constructeurs de matériel réseau sont de plus en plus conscients de l'importance grandissante du trafic IP/Multicast et des problèmes d'inondation qu'il entraîne sur un réseau de type Ethernet.

Pour faire face à ce problème (mais aussi à bien d'autres) certains de ces constructeurs (Cisco, Extreme Networks, Bay Networks, etc) proposent des pont filtrants "intelligents". Ces ponts filtrants sont intelligents car ils ne se contentent plus d'interpréter les adresses de la couche de liaison pour filtrer le trafic entre les différents sous-réseaux qu'ils interconnectent ; ils sont capables de reconnaître les informations des couches de protocole des niveaux supérieurs, jusqu'à la couche de niveau transport pour les plus performants d'entre eux. Ils peuvent ainsi utiliser ces informations pour permettre la mise en place de politiques de filtrage avancées.

L'élagage IGMP (*IGMP pruning*) est l'une de ces politique. Elle consiste à repérer et interpréter les requêtes du protocole IGMP, afin de déterminer les sous-réseaux sur lesquels les trames de type *multicast* qui contiennent les datagrammes du protocole IP/Multicast doivent être retransmises (et donc, afin d'éliminer les sous-réseaux sur lesquels elles n'ont pas besoin d'être envoyées).

Finalement, le *multi-unicast* est-il un si mauvais choix ? Comme nous l'avons vu, les solutions s'appuyant directement sur les couches de niveau liaison sont en général trop

---

1. Le composant DEC 21140, largement utilisé dans les adaptateurs FastEthernet actuels, divise l'espace d'adressage en 512 sous-espaces (dont les élément ne sont pas contigus) et utilise une fonction de hachage sur les adresses pour déterminer à quel sous-espace chaque adresse de *multicast* appartient.

contraignantes pour être envisagées sérieusement.

Sur un réseau local, le protocole IP/Multicast peut en revanche s'avérer intéressant, mais sous certaines conditions :

1. le réseau local doit posséder la capacité de multi-distribution ; dans le cas contraire, son efficacité est au mieux celle du multi-unicast, car IP/Multicast implique une phase d'enregistrement auprès des routeurs (IGMP) qui n'est pas absolument nécessaire, lorsque son utilisation se limite à un unique réseau local ;

2. la propagation du trafic IP/Multicast doit être contrôlée :

   – c'est le cas lorsque le réseau est clos ou non subdivisé (ce qui est courant dans les grappes de stations de travail, lorsque les stations sont attachées à deux réseaux locaux, l'un clos pour le trafic d'exploitation, et l'autre ouvert pour le trafic administratif) ;

   – c'est le cas lorsque le réseau est subdivisé à l'aide de routeurs ou des ponts filtrants intelligents.

Ajoutons toutefois que même lorsque ces conditions sont réunies, des facteurs autres que celui des performances, comme la sécurité (l'accès aux groupes de communication du protocole IP/Multicast n'est pas et ne peut pas être contrôlé), peuvent remettre en question l'intérêt du protocole IP/Multicast.

A la lumière de tous ces éléments, il apparaît donc clairement qu'un mécanisme de communication multipoints s'appuyant sur le *multi-unicast* est incontournable, car (i), il est indépendant du réseau sous-jacent (et de ses capacités) et (ii) il permet de mettre en place des mécanismes de sécurité. Lorsque le type et la configuration du réseau le permettent, et que les problèmes de sécurité peuvent être ignorés ou contournés (par le cryptage systématique des messages, par exemple), ce mécanisme peut éventuellement être suppléé par le mécanisme multipoints du protocole IP/Multicast. Or, comme nous l'avons vu au début de ce paragraphe, notre choix de recourir au protocole UDP pour la réalisation des communications point-à-point permet facilement d'envisager cette transition.

8.2.3 Présentation du protocole de niveau transport

De la même façon que TCP, le protocole de communication proposé utilise une technique d'accusés de réception positifs : les messages sont envoyés vers leur(s) destinataire(s) au moyen du mécanisme de communication point-à-point ou du mécanisme de diffusion. Lorsqu'un message est reçu, un acquittement est retourné à son expéditeur. Lorsque l'expéditeur s'aperçoit qu'un message qu'il a envoyé n'est pas acquitté par l'un de ses destinataires, ou lorsqu'un destinataire lui indique qu'un message semble s'être perdu, l'expéditeur retransmet le message vers ce destinataire, en utilisant cette fois le mécanisme de communication point-à-point.

Le protocole défini reprend en fait la majorité des techniques utilisées par le protocole TCP. Les accusés de réception sont incorporés aux messages de données quand il peuvent l'être (technique de *piggy-backing*) ou retardés dans le cas contraire (technique de *delayed-ack*). Chaque message est numéroté de façon séquentielle par son expéditeur. Cela permet d'une part au destinataire de les remettre en ordre lorsqu'il les reçoit, et d'autre part, de détecter la perte de certains messages. Lorsque l'acquittement d'un message n'est pas reçu après l'expiration d'un délai de garde, le message est automatiquement retransmis. De la même façon que dans TCP, ce délai de garde est calculé dynamiquement, à partir d'une estimation du temps d'aller-retour moyen des messages.

Le protocole met aussi en place un mécanisme de contrôle de flux et de congestion, basé sur le principe des fenêtres d'émission de tailles variables. Une fenêtre d'émission représente la quantité d'information qui peuvent être envoyées sans acquittement. Lorsque la fenêtre est entièrement utilisée, l'expéditeur stoppe ses émissions jusqu'à ce qu'un accusé de réception soit reçu. Lorsqu'un accusé de réception est reçu, les données acquittées sortent de la fenêtre, ce qui permet à l'expéditeur de reprendre ses émissions. La taille de la fenêtre est ajustée dynamiquement. Elle est (éventuellement) réduite lorsqu'un message est perdu et (éventuellement) agrandie lorsque des messages sont transmis consécutivement sans échec (algorithme *slow-start*). Le principe de fenétrage est toutefois moins complexe que celui de TCP, dont l'objectif est d'être capable de supporter les aléas du routage Internet. Comme sur un réseau local le routage est a priori statique, les fenêtres de tailles variables que nous utilisons indiquent le nombre de *messages* (datagrammes) qui peuvent être successivement envoyés sans être immédiatement acquittés.

8.2.4 Description des champs de l'en-tête réservés au niveau transport

Les messages MPCFS utilisent l'en-tête de taille fixe de (48 octets) qui est présenté sur la figure 8.2. Cet en-tête est partagé par les couches de protocole de niveau transport (champs représentés en gras) et session (champs représentés en italique). Le début de l'en-tête est plutôt réservé à la couche transport, mais pour des raisons d'alignement, certains champs de la couche session apparaissent au début de l'en-tête. Certains des champs de l'en-tête sont partagés par les deux couches de protocole (ils sont représentés en gras italique).

Les champs utilisés par la couche de niveau transport sont les suivants :

- **Identifiant MPCFS** : il s'agit d'une constante choisie arbitrairement pour identifier les datagrammes du protocole MPCFS. Cette constante permet (i) d'éliminer les datagrammes envoyés par erreur sur l'un des ports de réception utilisé par le système MPCFS et (ii) de reconnaître les éventuelles futures versions du protocole ;

- **Longueur** : il s'agit de la longueur totale du message, en-tête compris ;

- **Drapeaux** : ce champ, qui est utilisé à la fois par la couche de niveau transport et la couche de niveau session, regroupe les indicateurs booléens (drapeaux) concernant le message. Le seul indicateur concernant la couche transport est l'indicateur de retransmission d'un message (`MH_FL_RETRANS`, `0x08`). Cet indicateur est positionné par la machine expéditrice quand elle n'a pu déplacer tous les messages depuis sa file de

0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

0	**Identifiant MPCFS (5BA0)**		**Longueur**	
4	***Drapeaux***	***Type***	*Erreur*	Réservé
8	**Séquence**		**Séquence Acquittée**	
12	**Essai**	**Essai Acquitté**	*Identifiant Données*	
16	**Identifiant connexion source**		**Identifiant connexion destination**	
20	**Clef d'Identification source**			
24	**Clef d'Identification destination**			
28	*Identifiant Association source*		*Génération Association source*	
32	*Identifiant Association destination*		*Génération Association destination*	
36	*Sous-groupe source*		*Génération Sous-groupe source*	
40	*Sous-groupe destination*		*Génération Sous-groupe destination*	
44	***Options***			
48	Données ...			

FIG. 8.2: *En-tête des messages MPCFS*

remise en ordre vers sa file de traitement. Le message dont la retransmission est demandée est celui qui a le numéro de séquence qui suit immédiatement le numéro de séquence acquitté dans ce message.

- **Type** : de même que le précédent, l'utilisation de ce champ est partagée par les couches des niveaux transport et session. Les deux seuls types de messages qui soient spécifiques à la couche transport sont les messages de type "message vide" (`MH_TY_VOID`), et "déconnexion" (`MH_TY_CLOSE`). Un message de type vide est généré lorsque la couche de niveau transport doit acquitter un message reçu ou demander une retransmission alors qu'aucun message de niveau session n'est en attente d'émission. En effet, rappelons qu'afin de réduire le nombre de messages échangés, la stratégie par défaut consiste à profiter des messages de niveau session pour embarquer ces informations (technique de *piggybacking*). Le message de déconnexion est envoyé lorsque la connexion n'est plus utilisée par aucun groupe de communication.

- **Séquence** : numéro de séquence du message ; tous les messages à l'exception des messages de type vide (`MH_TY_VOID`) sont associés à un numéro de séquence. Les messages de type vide n'ont pas besoin de numéro de séquence car (i) leur perte n'a pas de conséquence grave sur le protocole (au pire, elle retarde la retransmission d'un message perdu ou provoque la retransmission d'un message déjà reçu) et (ii) leur contenu est calculé au moment de leur émission, selon l'état de la connexion.

- **Séquence acquittée** : numéro de séquence du dernier message placé sur la file de traitement ;

- **Essai** : ce champ contient le compteur de retransmissions du message. Cette information est indispensable afin de distinguer les multiples transmission d'un même mes-

sage, lors du calcul du temps d'aller-retour des messages ;

- **Essai Acquitté** : la valeur du champ essai du dernier message placé sur la file de traitement. Notons qu'afin de ne pas fausser le calcul du temps d'aller-retour d'un message, le compteur de retransmission des messages qui ne peuvent être déplacés depuis la file de remise en ordre vers la file de traitement à la fin de l'étape de lecture (voir la description de l'algorithme de traitement d'une connexion active, présenté au paragraphe 7.4.2, page 181) est mis à zéro. Rappelons en effet que les messages sont conservés dans la file de remise en ordre tant qu'un message les précédant est manquant. Lorsqu'un message manquant est reçu, ce n'est nécessairement le numéro de séquence de ce message qui est acquitté, mais celui du dernier message en séquence, qui peut être arrivé depuis longtemps ;

- **Identifiant connexion source** : identifiant de connexion de la machine expéditrice ;

- **Identifiant connexion destination** : identifiant de connexion de la machine destinataire. Ces deux identifiants permettent de supporter le multiplexage sur un même port UDP des datagrammes provenant des connexions de plusieurs machines. Ils n'ont que peu d'utilité avec le type de connexion strictement point-à-point utilisé actuellement par le prototype MPCFS, qui attribue un port UDP différent à chacune d'entre elles. En revanche, ils nous permettent d'envisager facilement la transition vers le protocole IP/Multicast, qui exige une politique de gestion des numéros ports différente de celle que nous utilisons actuellement. En effet, avec IP/Multicast, chaque machine membre d'un groupe reçoit les messages émis par chacune des autres machines au travers d'un même numéro de port ou ensemble de numéros de port ;

- **Clef d'identification source** : cette clef permet de vérifier que l'identifiant de connexion source est valide ;

- **Clef d'identification destination** : cette clef permet de vérifier que l'identifiant de connexion destination est valide ;

- **Options** : Ce champ est prévu afin d'autoriser la transmission d'informations optionnelles concernant les couches de niveau transport et session. Il n'est pour l'instant utilisé que par la couche de niveau session, mais nous envisageons à court terme de l'utiliser au niveau transport, afin d'améliorer la gestion du contrôle de flux (en fournissant des information concernant l'occupation des files de messages, par exemple).

8.2.5 Algorithme de traitement d'un message reçu

L'algorithme 8.1 décrit la procédure de lecture et de traitement d'un datagramme depuis un *socket* de réception, au niveau de la couche transport. Cet algorithme commence par prélire l'en-tête du message en attente sur le *socket* de réception de la connexion. Ensuite, il décode cet en-tête (conversion des champs de l'en-tête du format de représentation réseau vers le format de représentation de la machine locale) et vérifie qu'il est bien formé.

Puis, il traite une éventuelle demande de retransmission, en replaçant le message demandé, s'il existe, sur la file d'émission immédiate de la connexion.

Après cela, il s'occupe du numéro de séquence acquitté : il met éventuellement à jour la moyenne et la variance du temps d'aller-retour des messages sur cette connexion et élimine tous les messages acquittés.

Si le message est un message de type vide, le traitement s'arrête là. Dans le cas contraire, si il reste suffisamment de place sur les files de réception, le message est lu en entier et placé sur la file de remise en ordre. Dans le cas contraire, il est éliminé.

8.3 Couche session

Les objectifs de la couche session sont d'une part, de permettre l'échange des messages de données des utilisateurs au sein des groupes et sous-groupes de communication du système de fichiers MPCFS, et d'autre part, d'assurer la gestion répartie de ces groupes et sous-groupes de communication.

Pour cela, MPCFS s'appuie sur quatre sous-protocoles (voir figure 8.1, page 195) :

- un protocole d'échange de messages dans les sous-groupes de communication et de gestion de l'espace de stockage associé ;
- un protocole d'association de groupes de communication entre machines ;
- un protocole de création de sous-groupe de communication ;
- un protocole d'échange des identifiants de sous-groupe, invoqué par les deux protocoles précédents.

8.3.1 Description des champs de l'en-tête réservés au niveau session

Ces protocoles utilisent les champs suivants de l'en-tête de message MPCFS, que nous avons présenté au paragraphe 8.2.4 (figure 8.2, page 203) :

– le champ **Drapeaux**, qui peut contenir les indicateurs binaires suivants :

> `MH_FL_REPLY` est présent lorsque le message est un message de réponse à un message précédent. Ces messages sont de longueur minimale et n'entraînent pas eux-mêmes l'envoi en sens inverse d'un message de réponse car leur rôle est d'indiquer le résultat du traitement d'un autre message ;
>
> `MH_FL_ERR` est présent dans un message de réponse lorsqu'une erreur s'est produite lors du traitement du message associé à cette réponse. Un message de réponse pour lequel ce drapeau est positionné est appelé un *message de réponse négative* et un message (de réponse) dans lequel il n'est pas positionné un *message de réponse positif*.
>
> `MH_FL_MULTI` est positionné lorsque le message est un *message de réponse multiple*. Une message de réponse multiple combine plusieurs messages de réponse en un seul. Seuls les messages de réponse positifs peuvent être combinés de cette façon. Ce drapeau est donc incompatible avec le précédent.

ALG. 8.1 Algorithme de traitement d'un message reçu au niveau transport

```
fonction traitement_en-tête(socket,connexion)
début
  en-tête←lire_sans_consommer(socket,taille_en-tête)
  erreur←décoder_vérifier_en-tête(en-tête)

  si (erreur ≠ OK) alors
    vider_socket(socket)
    retour erreur
  finsi

  si (MH_FL_RETRANS présent dans en-tête→drapeaux) alors
    traiter_retransmission(en-tête,connexion)
  finsi

  message_original←retrouver_message(en-tête→séquence_acquittée,connexion)
  si (message_original existe) alors
    si (en-tête→essai_acquitté = message_original→essai_courant) alors
      mise_à_jour_temps_aller_retour(message_original,connexion)
    finsi

    tant que (premier_liste(connexion→file_émission)→séquence ≤
              en-tête→séquence)
    faire
      supprimer_tête_liste(connexion→file_émission)
    fait
  finsi

  si (en-tête→type = MH_TY_VOID) alors
    vider_socket(socket)
    retour OK
  finsi

  si (place_sur_files_réception(connexion)) alors
    message←lire_en_consommant(socket,en-tête→longueur)
    insérer_file(connexion→remise_en_ordre,message)
  sinon
    vider_socket(socket)
    retour MESSAGE_ÉLIMINÉ
  finsi

  retour OK
fin
```

– le champ **Type**, qui peux prendre les valeurs suivantes :

  MH_TY_ACT lorsqu'il s'agit d'une requête d'association de groupe de communication ;

  MH_TY_TINFO lorsqu'il s'agit de l'annonce d'une création de sous-groupe ;

  MH_TY_TTREE lorsqu'il s'agit d'un message du protocole d'échange des identifiants de sous-groupe ;

  MH_TY_DATA lorsqu'il s'agit d'un message de données ;

  MH_TY_TSTART lorsqu'il s'agit d'un message indiquant que les émissions qui avaient été interrompues dans un sous-groupe peuvent reprendre.

– le champ **Erreur** n'a de signification que lorsque le drapeau MH_FL_ERR est positionné dans le champs **Drapeaux** d'un message de réponse. Il peut prendre les valeurs suivantes :

  MH_ER_AAUTH indique que l'identifiant et/ou la génération d'une association figurant dans un message précédent sont erronés ;

  MH_ER_TAUTH indique que l'identifiant et/ou la génération d'un sous-groupe figurant dans un message précédent sont erronés ;

  MH_ER_GROUP indique que l'association de groupe demandée est impossible car le groupe n'existe pas ;

  MH_ER_TAG indique que l'annonce de création d'un sous-groupe à laquelle ce message répond est classée sans suite car le sous-groupe annoncé n'existe pas ;

  MH_ER_TSTOPA indique qu'un message de donnée reçu a pu être stocké, mais que les émissions de message de données doivent cesser jusqu'à nouvel ordre, car le répertoire de stockage ne peut plus accepter de message supplémentaire ;

  MH_ER_TSTOPR indique qu'un message de donnée reçu n'a pas pu être stocké et que (i) les émissions doivent cesser immédiatement jusqu'à nouvel ordre, et (ii) le message devra être retransmis.

– **Identifiant données** désigne un message de données dans un sous-groupe ;

– **Identifiant Association source** et **Génération Association source** désignent l'association permettant d'accéder au groupe de communication dans lequel le message est envoyé, sur la machine expéditrice ;

– **Identifiant Association destination** et **Génération Association destination** désignent l'association permettant d'accéder au groupe de communication dans lequel le message est envoyé, sur la machine destinataire ;

– **Sous-groupe source** et **Génération Sous-groupe source** désignent le sous-groupe dans lequel le message est envoyé, sur la machine expéditrice ;

– **Sous-groupe destination** et **Génération Sous-groupe destination** désignent le sous-groupe dans lequel le message est envoyé, sur la machine destinataire ;

– **Options** est prévu afin d'autoriser la transmission d'informations optionnelles. Au niveau session, la seule information optionnelle actuellement reconnue est l'identifiant du dernier message acquitté dans une réponse multiple (le premier message étant désigné par le champ **Identifiant Données**).

8.3.2 Protocole d'envoi des messages utilisateurs

La transmission des messages de données est confrontée au problème de l'espace de stockage limité dans les sous-groupes de communication. En effet, chaque message de données envoyé est susceptible d'être refusé par un ou plusieurs de ses destinataires. Comme le mécanisme de transmission est supposé fiable, chaque message refusé doit pouvoir être retransmis. Le protocole d'envoi des messages de données prévoir donc d'une part, un mécanisme d'acquittement pour indiquer si chaque message reçu a pu être accepté ou refusé, et d'autre part, un mécanisme de retransmission.

En ce qui concerne, le mécanisme d'acquittement, comme au niveau transport, nous avons opté pour un mécanisme d'acquittement positif. Dans le cas d'un flot de données unidirectionnel, le sur-coût de ce mécanisme d'acquittement est relativement limité car les acquittements des deux niveaux de protocole (session et transport) peuvent être combinés en un seul message de réponse. Dans le cas de flots de données bi-directionnels, ce double acquittement des messages est toutefois relativement coûteux, car sa version actuelle, notre protocole n'utilise pas encore la technique du *piggybacking*. Toutefois, afin de réduire ce sur-coût, nous avons élaboré un mécanisme d'acquittement multiple, grâce auquel plusieurs acquittements de niveau session peuvent être combinés en un seul message. Cette technique, qui n'est par ailleurs pas incompatible avec celle du *piggybacking* (que nous envisageons de mettre en place dans une version prochaine du protocole), offre des résultats d'autant meilleurs que les réseau et/ou le système deviennent surchargés.

En ce qui concerne le mécanisme de retransmission, nous avons opté pour un mécanisme dirigé par la destination. Lorsqu'un message est refusé par l'un de ses destinataires dans un sous-groupe, l'expéditeur cesse ses émissions dans ce sous-groupe vers ce destinataire, jusqu'à ce que le destinataire lui indique qu'il peut reprendre ses émissions. Ce mécanisme est plus avantageux qu'un mécanisme dirigé par la source, car le délai d'attente avant qu'un emplacement ne se libère au niveau du destinataire est imprévisible et potentiellement grand (un emplacement est libéré lorsqu'un message est consommé localement ou transmis et accepté par son destinataire).

Nous allons maintenant décrire plus en détail chacun des messages de ce protocole : les messages de données initiaux, les messages d'acceptation d'un message reçu (que nous appelons une *réponse positive*), les message d'acceptation de multiples messages de données reçus, les messages de refus et les message de reprise de émission lorsqu;elles sont interrompues à la suite d'un refus.

8.3.2.1 Envoi d'un message de données

L'envoi d'un message de données est déclenché à la suite d'une écriture sur l'un des fichiers `chan` de l'arborescence MPCFS. Au moment de cette écriture, chacune des associations qui existent dans le groupe sont parcourues afin de déterminer celles pour lesquelles le sous-groupe dans lequel le message est écrit possède un vis-à-vis. Le message est placé dans la file d'émission de chacune des connexion correspondant à ces associations. L'en-tête du message est initialisé juste avant que le message ne soit effectivement placé sur le *socket* d'émission : le champ **type** est initialisé à la valeur `MH_TY_DATA` et le champ **Identifiant Données** avec le numéro du message dans le répertoire de stockage local ; les **identifiants et numéros de génération de l'association et du sous-groupe sources et destinations** sont initialisés avec leurs valeurs respectives. Le champ **longueur** est initialisé avec la somme de la taille du message utilisateur (la quantité de données écrite sur le fichier `chan`) et de la taille de l'en-tête. Les champs **Drapeaux**, **Erreur** et **Options** ne sont pas utilisés.

8.3.2.2 Réponse positive à un message de données

Lorsque le message est accepté sans erreur, et qu'un message de réponse multiple ne peut être formé, un message de réponse simple est placé sur la file d'émission. Par défaut, ce message est formé à partir de l'en-tête du message reçu : le **Drapeau** `MH_FL_REPLY` est positionné, la **Longueur** est ramenée à la taille de l'en-tête et les identifiants et numéros de génération de l'association et du sous-groupe sont inversés (les sources deviennent les destinations et vice-versa).

8.3.2.3 Réponse positive à de multiples messages de données

Lorsque la file d'émission contient déjà un message de réponse positive, et qu'un deuxième message de réponse de positive doit être ajouté sur cette file, le protocole essaie de regrouper ces deux réponses en une seule.

Pour que deux réponses positives soit regroupées en une seule il faut :

1. qu'elles concernent les messages de données d'un même sous-groupe de communication ;
2. que ces messages soient consécutifs (selon l'ordre d'émission dans le sous-groupe) ; en d'autre termes, deux réponses ne peuvent être combinées si une réponse négative pour le même sous-groupe les sépare dans la file d'émission ;
3. que les réponses correspondantes n'aient pas déjà été envoyées ;

Dans ce cas, le message le plus récent est éliminé de la file (en fait il n'y est jamais placé) et le message le plus ancien est transformé en réponse multiple : le **Drapeau** `MH_FL_MULTI` de ce message est positionné et les 16 bits de poids fort de son champ `Options` sont initialisés avec l'**Identifiant de Données** du message le plus récent.

Lorsque la file d'émission contient déjà un message de réponse multiple et qu'un autre message de réponse positive doit être ajouté sur cette file, le protocole essaie de regrouper les

deux messages en un seul, suivant les même règles que précédemment (le champ Options du message de réponse multiple déjà présent sur la file est mis à jour avec l'**Identifiant de Données** du nouveau message).

Notons que ces regroupements sont d'autant plus profitables que le système et/ou le réseau sont surchargés. En effet, lorsque la charge augmente, les files d'émission se remplissent, soit parce que les fenêtres de transmission sont réduites (charge réseau), soit parce que le débit d'arrivée des messages sur la file d'émission est important (charge système). Or, justement, plus les files d'émission sont pleines, plus les chances de trouver un message de réponse en attente d'une première émission sont grandes, et plus le nombre de messages de réponse envoyés diminue.

8.3.2.4 Réponse négative à un message de données

Une réponse négative à un message de données est formée lorsque le traitement d'un message de donnée aboutit à une erreur. Les champs **Longueur**, **Type**, **Identifiants et numéros de Génération sources et destination de l'association et du sous-groupe** de l'en-tête sont initialisés de la même façon que pour une réponse positive. Les bits MH_FL_ERR et MH_FL_REPLY sont positionnés dans le champ **Drapeau** et le champ **Erreur** est initialisé en fonction de l'erreur.

8.3.2.5 Message de reprise des émissions

Lorsque des machines sont bloquées dans l'attente d'un emplacement libre dans un répertoire de stockage, et qu'un emplacement se libère, un message de reprise est envoyé à l'une des machines bloquées. Les machines destinataires de ces messages sont choisies de façon à garantir à chaque machine bloquée une chance égale de bénéficier d'un emplacement libre (les emplacements libérés sont attribués cycliquement à chacune des machines).

Un message de reprise est formé d'un unique en-tête MPCFS. Le champ **Type** est initialisé avec la valeur MH_TY_TSTART, et les champs concernant les associations et le sous-groupe sont initialisés en fonction de la machine choisie et du sous-groupe dans lequel l'emplacement s'est libéré.

8.3.3 Protocole d'association de groupes

Le protocole d'association des groupes de communication de deux machines est déclenché à l'initiative de l'une des deux machines, que nous appelons la machine *instigatrice*. Rappelons que ce protocole suppose qu'une connexion point-à-point a été établie au préalable entre les deux machines (voir l'algorithme d'établissement d'une connexion au paragraphe 7.4.5, page 189).

Comme nous l'avons vu au chapitre 7.4.3 (page 185), ce protocole procède en deux phases :

1. La machine instigatrice envoie un message de type MH_TY_ACT. En plus du champ **Type**, seuls les champs **Identifiant association source**, **Génération Association**

source et **Longueur** de l'en-tête de ce message sont initialisés. Le nom du groupe de communication est lui placé dans le corps du message. En réception de ce message, l'autre machine vérifie si elle possède un groupe dont le nom correspond à celui qui est placé dans le corps du message.

Si c'est le cas, elle forme un message réponse à partir de l'en-tête du message reçu : le les champs concernant l'association source sont copiés dans les champs destination équivalents, puis les champs association source sont initialisés avec les valeurs de cette machine ; le **Drapeau** `MH_FL_REPLY` est positionné et la **Longueur** initialisée avec la longueur de l'en-tête. Puis le message de réponse est placé sur la file d'émission. À partir de cet instant, l'association est opérationnelle pour cette machine, qui peut commencer à envoyer les messages au niveau du sous-groupe racine du groupe.

Si ce n'est pas le cas, elle forme un message d'erreur à partir de l'en-tête reçu : les champs concernant l'association sont inversés, les **Drapeaux** `MH_FL_REPLY` et `MH_FL_ERR` sont positionnés, le champ **Erreur** est initialisé à `MH_ER_GROUP` et la longueur est ramenée à la taille de l'en-tête.

2. Quand elle reçoit une réponse positive à son message, la machine instigatrice déclenche le protocole de calcul d'intersection d'arborescence réparties, présenté au paragraphe 8.3.5. L'association des deux groupes devient opérationnelle pour cette machine à partir de cet instant.

8.3.4 Protocole de création de sous-groupes

Rappelons qu'un sous-groupe est créé lorsqu'un premier processus crée un répertoire de multiplexage pour ce sous-groupe dans l'arborescence propre d'un processus membre du groupe. Le protocole déclenché par cette opération procède en deux étapes :

1. Un message d'annonce de la création du sous-groupe est diffusé dans le sous-groupe parent. Ce message est donc envoyé vers chacune des machines associée au groupe de communication qui définit le sous-groupe parent. L'en-tête de message d'annonce est initialisé de la façon suivante : le champ **Type** prend la valeur `MH_TY_TINFO`, les champs décrivant l'association sont initialisés en fonction des destinataires du message et les champs concernant le sous-groupe sont initialisés avec les valeur du sous-groupe parent. Le corps du message est initialisé avec une structure décrivant le nouveau sous-groupe. Cette structure contient les informations suivantes :

   – le nom du nouveau sous-groupe ;

   – l'identifiant du nouveau sous-groupe ;

   – le numéro de génération du nouveau sous-groupe.

2. La réception d'un tel message ne provoque jamais l'envoi d'un message de réponse. Si le sous-groupe n'existe pas sur la machine qui reçoit ce message, le message est simplement ignoré. Au contraire, si le sous-groupe existe, la machine déclenche le protocole de calcul d'intersection d'arborescence réparties avec la machine ayant envoyé le message d'annonce, à partir de ce nouveau sous-groupe commun. Ce mode

de fonctionnement permet au protocole de fonctionner lorsque plusieurs sous-groupes imbriqués sont crées en un laps de temps très court.

Supposons par exemple que les sous-groupes “/a” et “/a/b”, soient successivement créés en un laps de temps très court sur une machine A. Supposons aussi que ces deux sous-groupes existent aussi sur une machine B à laquelle A est associée. La création du sous-groupe “/a” entraîne bien l’envoi d’un message d’annonce depuis la machine A vers la machine B. Mais tant que la machine B n’a pas transmis ses propres identifiants pour ce ce sous-groupe à la machine A, l’association de ce sous-groupe entre les deux machines n’est pas opérationnelle. En conséquence, la machine A ne peut pas envoyer le deuxième message d’annonce concernant la création du sous-groupe “/a/b”, bien qu’il existe sur les deux machines. On voit donc clairement que pour ne pas oublier de sous-groupes dans une telle situation, il faut explorer systématiquement les sous-arborescence de sous-groupes à chaque fois qu’un sous-groupe est créé, ce que fait le protocole de calcul d’intersection d’arborescences, décrit au paragraphe 8.3.5.

8.3.5 Calcul d’intersection d’arborescences réparties

La vocation des sous-groupes de communication est d’assurer le multiplexage des flots de communication au sein d’un groupe de communication. Ce multiplexage est obtenu en étiquetant chaque message, en fonction du chemin emprunté dans l’arborescence des sous-groupes pour atteindre le nœud de communication utilisé pour envoyer ou recevoir le message.

8.3.5.1 Présentation du problème

Pour des raisons évidentes de performance, ce chemin, qui peut être arbitrairement long, ne peut pas être directement attaché à chacun des messages transmis. Au lieu de cela, ce sont les identifiants du groupe de communication et de sous-groupe dans le groupe qui sont utilisés afin d’étiqueter les messages.

Or, le mode de fonctionnement totalement réparti du système MPCFS implique que chaque machine gère de façon indépendante sa propre arborescence de sous-groupes, au sein de chacun des groupes de communication. Cette arborescence évolue dynamiquement, au gré des créations ou destructions de répertoires opérées par les processus qui s’exécutent sur chacune des machines. En conséquence, chaque machine attribue librement (c’est-à-dire de façon non concertée) ses propres identifiants à chacun des sous-groupes de ses groupes de communication[2].

Le problème relativement classique que l’on doit résoudre, est de reconnaître et interpréter de correctement les identifiants des autres machines. Pour cela, nous n’avons d’autre alternative que de mettre en place un *protocole d’échange des identifiants locaux*. Autrement dit, chaque machine doit, à un moment donné, échanger des messages du type : “pour moi,

2. À l’exception toutefois du sous-groupe racine d’un groupe dont l’identifiant est une valeur fixée de façon statique (le sous-groupe racine est créé automatiquement à la création du groupe, et ne peut par la suite être détruit).

machine A, l'identifiant du sous-groupe de i-ème génération correspondant au chemin /a/b/c dans le groupe G est I" (par la suite, un tel message est appelé une *annonce*).

L'efficacité de ce protocole d'échange est déterminante pour assurer de performances globales du système. Idéalement, ce protocole ne doit pas introduire de latence supplémentaire dans la transmission des messages utilisateurs, de plus il doit préserver la bande passante utile du réseau.

Ne pas introduire de latence supplémentaire implique disposer de l'information utile au moment de la transmission des messages en ayant recours à *un protocole à conservation d'état*. Nous avons, pour cela, choisi de construire sur chaque machine un ensemble de tables associatives (une par machine et par groupe), permettant la conversion directe des identifiants locaux en leurs équivalents sur les autres machines.

Préserver la bande passante implique de minimiser le nombre et le volume des messages de contrôle échangés. ceci est réalisé lorsque seules les informations utiles sont échangées. Dans notre cas, les informations utiles reçues par une machine en provenance d'une autre machine sont les annonces qui concernent un sous-groupe qui existe simultanément sur les deux machines. En d'autres termes, il s'agit des annonces qui concernent les sous-groupes appartenant à l'*intersection des arborescences* de sous-groupes des deux machines.

8.3.5.2 Description du protocole

Pour chacune des deux machines impliquées, l'objectif de ce protocole est de construire *l'intersection* des arborescences de sous-groupes d'un de leur groupe commun, et d'associer les identificateurs représentant le même sous-groupe.

Il faut aussi s'assurer que des parties de l'arborescence ne sont pas oubliées, sachant que cette arborescence évolue dynamiquement, au gré des créations et destructions de groupes par les processus.

Dans la description qui suit, nous appelons *étoile* un sous-arbre de profondeur au plus 1, c'est-à-dire soit un nœud seul, soit un nœud et ses fils directs.

Le protocole élaboré explore l'arborescence en largeur d'abord depuis la racine du sous-arbre choisi (un exemple d'exécution est présenté figure 8.3). L'exploration s'interrompt lorsque le protocole atteint un niveau de l'arborescence ne contenant plus aucun sous-groupe commun aux deux machines (ou que tous les niveaux de l'arborescence ont été couverts). Chacun des messages de type *Arbre* utilisé peut contenir plusieurs descriptions d'étoiles, en restant dans la limite de la taille maximale de message autorisée. Lorsqu'un message est plein mais que le niveau courant n'est pas totalement exploré, un message supplémentaire est utilisé, et le précédent envoyé. Chacune des étoiles dont la description est placée dans un message est marquée afin de ne pas être reparcourue (ce qui aurait des conséquences catastrophiques dans le cas de créations en série de sous-groupes d'un groupe ayant une arborescence importante). Une description d'étoile est formée des identificateurs source et destination de la racine de l'étoile ainsi que de la liste des noms et identificateurs source des feuilles de l'étoile non encore marquées.

La machine qui déclenche le protocole commence par former un message ne contenant que la description de l'étoile racine. Ensuite, lors de chaque réception d'un message de type **Arbre**, la machine destinataire extrait les descriptions des étoiles du message reçu et procède

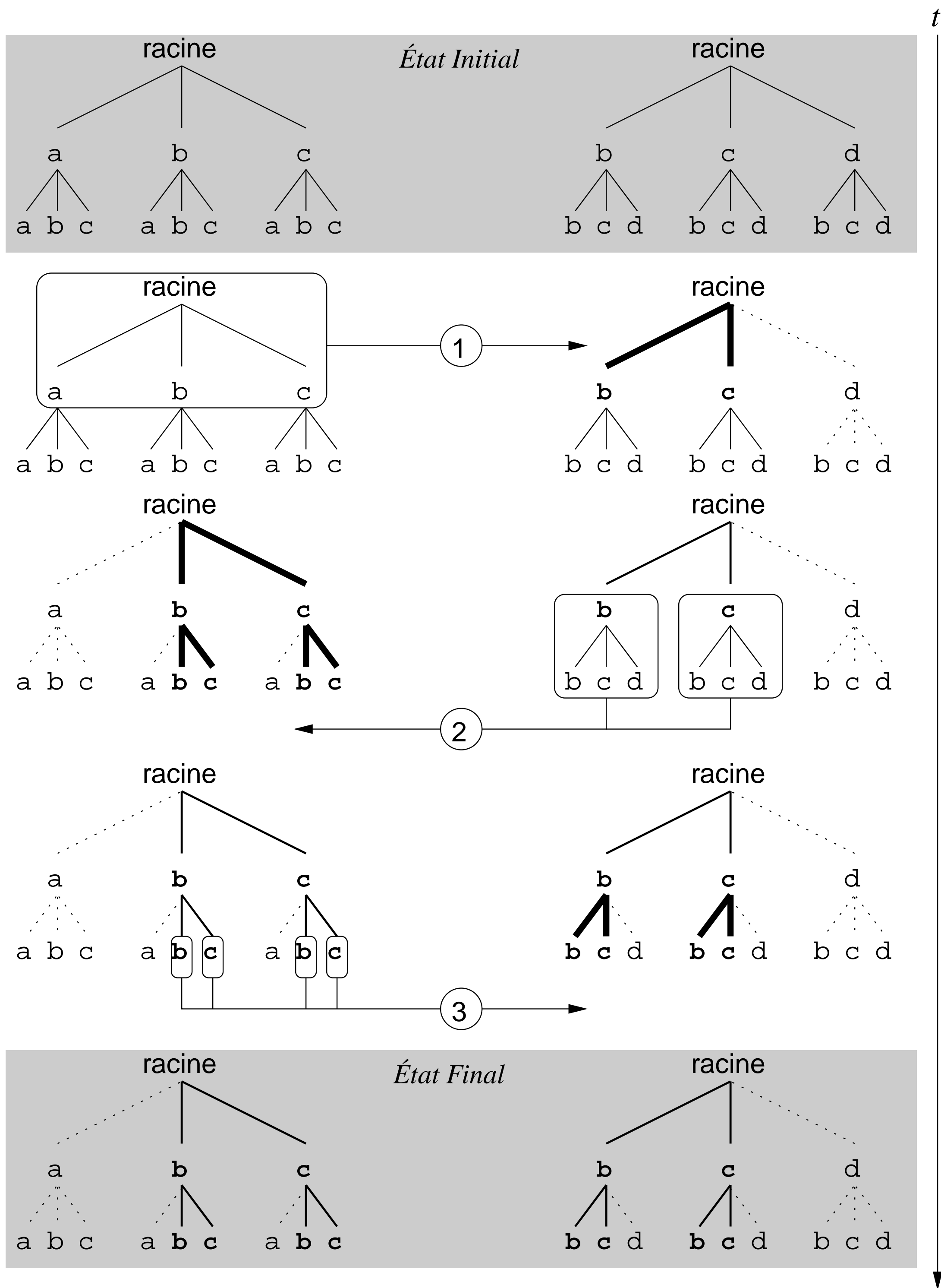


FIG. 8.3: *Exemple de calcul réparti d'intersection d'arborescence*

pour chacune d'elle de la façon suivante :

1. l'identificateur source de la racine, a priori encore inconnu, est sauvegardé ;

2. la liste des feuilles de l'étoile reçue est parcourue afin de rechercher les nœuds communs non encore atteints ; pour chacun de ces derniers, la description de l'étoile dont ils sont racine est ajoutée au message réponse de type **Arbre** en cours de remplissage.

Lorsque tous les étoiles reçues ont été traitées, le dernier message de type **Arbre** qui était en cours de remplissage est envoyé.

8.3.5.3 Analyse de la complexité du protocole

Nous analysons le protocole dans le cas où les arbres initiaux sont équilibrés et de même caractéristiques. Nous allons montrer que ce protocole est efficace en nombre de messages échangés et en volume d'informations transmises.

Soient

– d, le degré (nombre de fils par nœud) ;

– L, la profondeur des arbres initiaux sur chacune des deux machines ;

– d_i et L_i le degré et la profondeur de leur intersection ;

– T_d (resp. T_i) la taille d'un descripteur de nœud (resp. d'un identifiant) ;

– T_m la taille maximale d'un message.

Nous notons $\mathcal{C}(L_i, d_i)$ (resp. $\mathcal{A}(L_i, d_i)$) le nombre de messages utilisés par un protocole optimal (resp. par notre protocole).

La quantité minimale d'information qui doit être transmise afin de calculer le p-ième niveau de l'arbre intersection est $q_p = d_i^{p-1} \cdot d$ descriptions de nœuds. Les nœuds fils d'un nœud faisant partie de l'arbre intersection au niveau $p-1$ sur l'une des deux machines doivent être comparés aux nœuds fils de ce même nœud de niveau $p-1$ sur l'autre machine. Le niveau $p-1$ de l'arbre intersection comporte d_i^{p-1} nœuds, ayant chacun d fils. Posons $Q_p = q_p \cdot T_d$; la quantité minimale d'informations qui doit être échangée par un protocole calculant l'arbre intersection est alors :

$$\sum_{p=1}^{L_i} Q_p$$

Ceci implique que

$$\mathcal{C}(L_i, d_i) \geq \left\lceil \frac{\sum_{p=1}^{L_i} Q_p}{T_m} \right\rceil = \left\lceil d \cdot \frac{1 - d_i^{L_i+1}}{1 - d_i} \cdot \frac{T_d}{T_m} \right\rceil$$

Notre protocole, ne compare, à chaque étape, que les nœuds fils des nœuds de l'arbre intersection obtenus à l'étape précédente. Il est donc efficace de ce point de vue : seuls les identifiants des nœuds retenus à l'étape p sont "inutilement" répétés dans les messages transmis à l'étape $p+1$, afin d'identifier les racines des étoiles transmises. Lors de l'étape p, la quantité d'information transmise par notre protocole est de :

- $d \cdot d^{p-1} T_d$ pour les descripteurs de nœuds;
- $d_i^{p-1} T_i$ pour les identifiants.

Comme $d_i^{p-1} < d \cdot d_i^{p-1}$, et que $T_i << T_d$, nous négligeons le second terme. Ainsi, a chaque étape, notre protocole transmet $d \cdot d_i^{p-1} T_d = Q_p$ informations, soit $\left\lceil \frac{Q_p}{T_m} \right\rceil$ messages. Ainsi $\mathcal{A}(L_i, d_i) \leq \sum_{i=1}^{i=L_i} \left\lceil \frac{Q_p}{T_m} \right\rceil$. Au total nous avons $\mathcal{A}(L_i, d_i) \leq \mathcal{C}(L_i, d_i) + L_i - 1$. et notre protocole diverge de l'optimal d'au plus $L_i - 1$ messages.

Le protocole que nous proposons est donc relativement efficace. En pratique, il semble en fait difficile d'améliorer ce résultat.

Par exemple, on peut envisager de diminuer le nombre d'étapes afin de diminuer le nombre des messages échangés. Mais cela impliquerait d'augmenter la profondeur des sous-arbres échangés à chacune des étapes, avec pour conséquence, la transmission inutile de certaines parties des arborescences. De façon générale, la complexité en nombre d'étapes de ce protocole lorsque les sous-arbres échangés sont de profondeur k est en effet de l'ordre de $\lceil \frac{L_i}{k} \rceil$ alors que la quantité d'informations transmises à chaque étape croît exponentiellement en d^k, avec la profondeur des sous-arbres. Et cette quantité d'informations diverge d'autant plus de la quantité optimale que la différence entre d et d_i est importante (l'optimal croît en d_i^k). Donc si diminuer le nombre d'étapes permet dans certains cas particuliers de gagner quelques messages (par exemple, lorsque d et d_i sont égaux), cela se traduit dans le cas général par une augmentation conséquente de la quantité d'information transmises et, par conséquent, du nombre de messages requis. Quant à augmenter le nombre d'étapes, cela ne ferait qu'augmenter le nombre de messages requis, sans diminuer la quantité d'informations transmises, qui est déjà pratiquement minimale.

8.4 Conclusion

Dans ce chapitre, nous avons présenté les deux couches de protocoles développées pour le prototype MPCFS : la couche de niveau transport, qui assure la transmission fiable et ordonnée selon la source des messages entre machines, et la couche de niveau session, qui assure la prise en charge et la transmission des messages au niveau utilisateur, ainsi que la gestion répartie des groupes et sous-groupes de communication.

Au niveau le plus bas, la couche de niveau transport s'appuie sur le procole UDP/IP, ce qui garantit au système une grande portabilité. Le prototype n'exploite pas, actuellement, les capacités du multi-distribution du protocole IP/Multicast, mais le choix du protocole UDP/IP nous permet facilement d'envisager cette transition. Ce protocole de niveau transport est fortement inspiré du protocole TCP/IP, dont il reprend bon nombre des mécanismes de contrôle

de flux et d'évitement de congestion. Précisons que notre objectif avec ce protocole n'est pas de rechercher une solution originale au difficile problème du multicast fiable et ordonné, mais seulement de fournir les mécanismes de base qui permettent au reste du prototype de fonctionner. Par la suite, nous envisageons l'adaptation de protocoles plus élaborés, utilisant des techniques d'acquitements négatifs telles que celle du protocole RAMP [Braudes 93].

Au niveau le plus haut, nous avons présenté les quatre sous-protocoles de niveau session développés pour le prototype MPCFS : protocole de prise en charge des messages de données des utilisateurs, le protocole d'association des groupes de communication, le protocole de création des sous-groupes de communication et le protocole de calcul de l'intersection d'arborescences réparties. Pour ce dernier, qui pose un problème d'algorithmique répartie original, nous avons de plus donné une analyse détaillée de la complexité de la solution proposée.

Les protocoles présentés sont opérationnels, mais n'ont pour l'instant été testés que sur une petite plate-forme, composée de trois PC fonctionnant sous le système Linux 2.0.34. Les performances mesurées sur cette plate-forme lors de nos premiers tests sont encourageantes, avec un débit point à point atteignant 9 MBits/s utiles sur un réseau Ethernet à 10 MBits/s (pour des messages de 10 kilo-octets). En revanche, ces performances se dégradent sensiblement pour des communications multi-points, avec un débit cumulé maximal de 7,5 MBits/s pour des messages de 10 kilo-octets. Le prototype actuel de MPCFS comporte néanmoins de nombreuses instructions de mise au point, dont la suppression nous permettrait, à terme, d'envisager un gain appréciable de performance.

Chapitre 9

Conclusions et Perspectives

LA puissance de calcul, et d'une façon générale, l'ensemble des ressources offertes par les réseaux de stations de travail, leur grande disponibilité et leur faible coût, suscitent, depuis le début des années 90, l'intérêt d'une communauté toujours plus grande d'utilisateurs. Cet intérêt se traduit par une demande croissante d'outils tant pour la programmation que pour le support d'exécution des applications réparties. Pour faire face à ce nouveau besoin, les premiers outils développés se sont contentés de composer des solutions logicielles à partir des outils et mécanismes existants, et/ou d'adapter des stratégies développées dans d'autres contextes, comme les techniques de programmation séquentielles ou les techniques de programmation employées sur les super-calculateurs parallèles. Aujourd'hui, les limites de ces techniques et outils apparaissent plus clairement. Les mécanismes de communication utilisés sont inadéquats. La forte hétérogénéité de ces plate-formes est supportée, mais rarement exploitée. Le contexte multi-utilisateurs et multi-tâches est mal géré, voire simplement ignoré. L'évolution rapide des configurations est difficilement prise en compte. Etc.

Dans cette thèse, nous nous sommes intéressés à deux problèmes importants qui se posent lorsque l'on utilise les réseaux de stations de travail, afin d'exécuter des applications parallèles et réparties : le problème de la *répartition dynamique de charge* et celui des *communications multipoints*. Pour résoudre chacun de ces deux problèmes, nous proposons deux outils originaux.

En ce qui concerne la répartition dynamique de charge, il s'agit de *LoadBuilder*, un environnement réparti spécifiquement conçu pour la construction de plans d'expériences sur les réseaux de stations de travail. Le but de ces plans d'expériences est de modéliser le comportement des machines d'un réseau de stations de travail, en fonction de leurs niveaux d'activité (charge). Cette modélisation a pour objectif de faciliter la mise en œuvre de politiques de répartition dynamique de charge dans l'environnement multi-tâches, multi-utilisateurs et fortement hétérogène des réseaux de stations de travail. L'environnement, de type client/serveur, permet la génération automatique d'expériences réparties, leur contrôle, et la récupération des données qu'elles génèrent. Lors de l'arrivée d'une nouvelle tâche dans le système, ces informations peuvent ensuite être utilisées afin de déterminer la machine qui, potentiellement, exécutera cette tâche le plus rapidement.

Pour ce qui est des communications multipoint, il s'agit de MPCFS (Multi-Point Communication File System), un module d'extension (pilote de périphérique) pour les systèmes UNIX qui implémente un système de fichiers virtuel réparti. Ce pilote de périphérique, qui s'exécute directement dans le noyau du système UNIX, permet la réalisation de communications multipoints fiables et ordonnées, grâce aux opérations classiques de manipulation de fichiers et de répertoires. Cette intégration au noyau du système permet, d'un côté, d'assurer la facilité et la transparence d'utilisation des mécanismes de communication, et de l'autre, de permettre d'atteindre les meilleurs niveaux de performance.

9.1 Construction d'indicateurs de charge

Pour répartir dynamiquement la charge de travail d'un ensemble de machines, il faut, dans un premier temps, savoir évaluer cette charge sur chacune des machines. Meilleure est cette évaluation, plus on est en droit d'espérer de bons résultats du mécanisme de répartition de charge. Or, l'évaluation de cette charge un problème difficile. En effet, le problème qui se pose n'est pas seulement celui de la mesure de la charge, mais aussi celui de déterminer quelle charge doit être mesurée. Paradoxalement, bon nombre de travaux concernant la répartition dynamique de charge n'accordent qu'une importance toute relative à ce problème, en préfèrant se concentrer sur la façon dont les informations de charge doivent être utilisées. Certes, une mauvaise exploitation de l'information ne conduit pas au meilleur résultat. Mais l'exploitation de mauvaises informations, aussi subtile soit elle, ne saurait non plus y parvenir.

Bien sûr, des méthodes d'évaluation de la charge existent et ont même, pour certaines, fait l'objet d'études approfondies, comme [Ferrari 86]. Nous avons recensé les principales de ces méthodes au chapitre 2. Toutefois, ces méthodes ont été élaborées dans des contextes différents, c'est-à-dire pour répondre à des besoins différents.

Ainsi, la répartition de la charge d'applications séquentielles, sur les réseaux de stations de travail, a initialement conduit à l'élaboration d'indicateurs mono-dimension et le plus souvent mono-critères (c'est-à-dire n'observant que la charge de travail du processeur), en contexte multi-utilisateur. L'hétérogénéité de la configuration pouvait alors être gérée manuellement, par le calcul d'un unique facteur, dont le rôle était de corriger la différence de puissance des machines (ou plus exactement, de leur processeur).

Puis, la répartition de la charge d'applications parallèles, sur les machines parallèles, a conduit à l'élaboration d'indicateurs multi-critères et multi-dimensions, mais dans un contexte homogène et souvent mono-utilisateur.

Aujourd'hui, le problème de répartition dynamique que l'on cherche à résoudre est une combinaison des difficultés rencontrées dans les contextes précédents, puisqu'il s'agit de répartir la charge d'applications parallèles et réparties sur des réseaux de stations de travail. Les indicateurs dont necéssaires dans ce contexte doivent donc être multi-dimensions, multi-critères, et tenir compte à la fois, de la nature multi-utilisateurs et multi-tâches de l'architecture, et de son hétérogénéité.

Les premiers travaux sur ce problème se sont inspirés des techniques proposées dans les contextes sus-cités, pour élaborer des indicateurs de charge multi-critères et multi-dimensions, qui soient *compatibles* avec l'hétérogénéité des réseaux de stations de travail [Folliot 92, Bernon 95, Chatonnay 98]. Toutefois, ces solutions ne cherchent pas encore à *exploiter* cette hétérogénéité. Par exemple, aucune des solutions proposées ne permet, à notre connaissance, d'exploiter les différences de performance des machines selon la nature des traitements demandés : calculs en nombre flottants ou en nombres entiers, utilisation intensive de la mémoire sur des zones restreintes ou étendues, etc.

En effet, exploiter cette hétérogénéité signifie qu'il faut construire des modèles du comportement des machines et du réseau qui intègrent chacune de leurs spécificités. Ces modèles comportent donc de très nombreux paramètres, dont l'évaluation ne peut être faite que de façon empirique, au travers de plans d'expériences. Cette évaluation, qui de plus doit être

reconduite à chaque fois qu'un changement a lieu dans la configuration, ne peut donc plus être réalisée manuellement.

Dans cette thèse, notre contribution à la résolution de ce problème est présentée au chapitre 4. Dans ce chapitre, nous proposons une méthodologie pour la construction automatisée des indicateurs de charge et nous présentons la plate-forme d'expérimentation que nous avons développée afin de la mettre en œuvre. Cette plate-forme permet notamment de construire des niveaux de charge synthétiques et d'en étudier les conséquences sur le fonctionnement des machines. L'observation se fait d'une part, au travers de la collecte des statistiques de fonctionnement proposées par les systèmes d'exploitation, et d'autre part, au travers de mesures de performances.

Toutefois, ces travaux ne constituent qu'une première étape. Cette plate-forme doit maintenant être exploitée, lors d'une seconde étape, pour construire l'ensemble des modèles recherchés et, lors d'une troisième étape, pour construire un outil permettant de calculer automatiquement les paramètres de ces modèles pour chacun des éléments d'un réseau de stations de travail.

Les travaux concernant la deuxième étape sont en cours, notamment avec la modélisation des performances des communications sur un réseau de stations de travail. Cette modélisation est une opération longue, au cours de laquel nous serons régulièrement amenés à corriger l'environnement *LoadBuilder*.

En effet, la définition des modules de charge synthétique dans *LoadBuilder* est fortement dépendante des modèles qui sont construits. Les modules de charge synthétique qui sont présentés devront donc évoluer en conséquence. Parmi ces évolutions, celles que nous envisageons dans un avenir proche sont d'une part, l'ajoût d'un module de simulation de la charge des entrées/sorties disque, et, d'autre part, l'ajoût de modules qui mêlent simultanément plusieurs types de charge synthétique. En effet, notre choix de recourir à des processus spécialisés pour chaque type de charge nous a permis, dans un premier temps, de simplifier le développement de la plate-forme *LoadBuilder* en la faisant évoluer de façon incrémentale, de déterminer l'ensemble des routines élémentaires qui peuvent être utilisées afin de simuler la charge, et de détecter les indicateurs du système qui sont sensibles à ces niveaux de charge. Cependant, les processus de *LoadBuilder* utilisent de façon régulière les mécanismes d'attente et de signalisation du système d'exploitation. La combinaison d'un nombre conséquent de ces processus afin de construire des niveaux de charge complexes entraîne donc systématiquement l'apparition d'une charge parasite, qu'il nous faut maintenant chercher à minimiser.

Les études préliminaires concernant la troisième étape sont elles aussi en cours. Nous envisageons en particulier d'utiliser le système CLIP/CLASP développé à l'Université du Massachussetts [Anderson 94]. Ce système exige toutefois un important travail d'adaptation, car il a été développé au-dessus d'un environnement Common Lisp propriétaire, qui n'est pas exploitable sans l'acquisition d'une licence. Dans un premier temps, le travail d'adaptation requis consiste à porter les parties du système CLIP/CLASP dont nous avons besoin au-dessus d'un environnement Common Lisp du domaine public. Dans un deuxième temps, il nous faut réaliser une interface entre ce système de pilotage d'expériences et la plate-forme

LoadBuilder. Enfin, il nous faut bien sûr élaborer les plans d'expérience recherchés et définir les procédures de calibration automatique des machines.

9.2 Mécanismes de communication multipoints

Les tâches des applications parallèles et réparties ont souvent recours à des schémas de communication multipoints, que ce soit pour communiquer entre-elles, ou simplement pour se synchroniser. Malheureusement, les systèmes d'exploitation disponibles sur les réseaux de stations de travail, et en particulier UNIX, n'ont pas été conçus pour offrir ce type de service.

Pour compenser ce manque, les solutions proposées jusqu'ici, que nous recensons au chapitre 3, consistent soit à définir un environnement de programmation formé de bibliothèques et de processus de service, soit à construire un nouveau système d'exploitation, soit à proposer un accès à des protocoles de communication multipoints au travers, par exemple, des APIs de communication point-à-point existantes. Chacune de ces catégories de solutions a ses avantages et ses inconvénients :

– l'avantage des environnements de programmation, tels que PVM [Geist 94], est qu'ils permettent de proposer des fonctionnalités avancées aux applications, en se satisfaisant du système d'exploitation disponible. Mais, en contre-partie, ils ne gèrent que très difficilement la concurrence entre plusieurs applications et sont limités sur le plan des performances (car ils ne peuvent recourir qu'aux fonctionnlités disponibles, qui sont essentiellement de type point-à-point) ;

– l'avantage des systèmes répartis, tels que Amoeba [Mullender 90], est leur efficacité : non seulement ils proposent aux applications des fonctionnalités avancées, mais de plus celles-ci peuvent être exécutées depuis le contexte d'exécution du système d'exploitation. Ils permettent donc de gérer efficacement la concurrence entre applications et permettent d'atteindre des niveaux de performances optimaux.

  L'un des inconvénients de ces systèmes est qu'ils obligent à abandonner le système d'exploitation initial, au profit d'un autre système. Cela se traduit à la fois par des problèmes de portabilité et des problèmes d'ordre sociologique (imposer le changement du système d'exploitation utilisé sur un réseau de stations de travail n'est pas toujours possible).

– l'avantage des protocoles de communication multipoints, tels que IP/Multicast [Deering 89] ou XTP [XTP Forum 95], est qu'ils sont, en principe, indépendants du système d'exploitation et qu'ils peuvent permettre d'exploiter efficacement les capacités d'un réseau.

  En revanche, les protocoles ne résolvent au mieux que partiellement le problème des communications multipoints, car ils doivent être assistés par un mécanisme de communication qui réalise l'interface avec les applications. Ce mécanisme peut être l'un des deux précédents ou simplement l'adaptation du mécanisme de communication point-à-point existant. Par ailleurs, précisons qu'à ce jour, aucun protocole idéal (c'est-à-dire

capable de satisfaire tous les besoins en matière de communications multipoints) n'a encore émergé.

La solution que nous proposons pour répondre au problème des communications multipoints, MPCFS, combine la plupart des avantages des solutions existantes, tout en minimisant leurs inconvénients.

En effet, l'utilisation de ce système, que nous décrivons au chapitre 5, est relativement simple puisqu'elle s'appuie sur l'API des systèmes de fichiers UNIX. Elle n'exige donc pas de bibliothèques de communication spécifique et est compatible avec l'ensemble des programmes UNIX existants, au travers du mécanisme de redirection des descripteurs d'entrée/sortie standard des processus. Son organisation arborescente permet de proposer des fonctionnalités avancées, comme des groupes de communication hiérarchiques. Les permissions sur les fichiers et répertoires de l'arborescence permettent tant aux utilisateurs qu'à l'administrateur du système de régler de façon sélective l'accès aux fonctionnalités du système.

Par ailleurs, son implémentation, dans le contexte d'exécution du noyau UNIX, lui donne potentiellement accès aux mêmes niveaux de performances que les solutions proposées dans les systèmes répartis, et permet de supporter efficacement l'exécution simultanée de plusieurs applications, appartenant à des utilisateurs différents. Ce niveau d'implémentation permet, de plus, d'accéder de façon privilégiée aux périphériques de communication et à tous les niveaux de protocole, et ce, de façon totalement transparente pour l'utilisateur.

Nous avons développé un premier prototype du système MPCFS pour le système Linux 2.0. Son implémentation (environ 10000 lignes de code source [1]) est décrite dans les chapitres 6 à 8.

La description que nous donnons des algorithmes et protocoles mis en œuvre dans ce prototype nous a permis, en particulier, de démontrer que la mise en œuvre des fonctionnalités du système MPCFS n'est que très peu dépendante du système choisi. En effet, les algorithmes de la partie haute du système, c'est-à-dire situés à l'interface entre les utilisateurs et le système, ne dépendent que de la disponibilité d'un mécanisme de type VFS [Kleiman 86], aujourd'hui disponible sur la majorité des systèmes UNIX. La partie basse du système, qui réalise l'interface du système avec le réseau, ne s'appuie, quant à elle, que sur le mécanisme des *sockets* pour accéder au protocole UDP/IP, et sur des mécanismes classiques de gestion d'évènements, comme les files d'attente et les temporisateurs (*timers*). L'ensemble des fonctionnalités du noyau Linux que nous avons utilisé pour développer ce prototype est donc largement disponible sur la majorité des systèmes d'exploitation UNIX, ce qui devrait en permettre l'adaptation aux principaux systèmes UNIX actuels (Solaris, Digital UNIX, etc).

1. Après suppression des commentaires et lignes vides. Les sources bruts représentent environ 17000 lignes de code. La compilation du code source génère un code objet de 100 Ko, dont le chargement se traduit par une occupation minimale de 25 pages de mémoire dans le noyau Linux (sur une architecture Intel).

La version actuelle du prototype (0.6.6) intègre l'ensemble des fonctionnalités de communication et de gestion répartie des groupes de communication que nous avons décrites. En revanche, la gestion des permissions au travers des fichiers de configuration n'est pas encore assurée (tous les fichiers et répertoires sont donc créés avec les mêmes permissions).

Les protocoles de communication que nous avons developpés pour ce prototype sont construits au-dessus du protocole UDP/IP. Pour réaliser les communications multipoints, le prototype utilise pour l'instant la technique du *multi-unicast*, mais nous envisageons, dans un proche avenir, d'ajouter une interface avec le protocole IP/Multicast.

Outre la mise en œuvre des fonctionnalités précédentes, nos travaux sur le système MPCFS vont se poursuivre avec un important travail de génie logiciel. En effet, nos principales préoccupations jusqu'à présent étaient de rechercher des solutions pratiques à chacun des problèmes techniques rencontrés. Par conséquent, l'architecture logicielle du prototype est relativement monolithique. L'objectif de ce travail de génie logiciel est de rendre cette architecture plus modulaire afin, d'une part, de permettre au prototype de supporter simultanément plusieurs types et plusieurs sémantiques de protocoles multipoints et afin, d'autre part, de faciliter son adaptation à d'autres plate-formes.

9.3 Perspectives

Concernant la répartition de charge, comme nous l'avons indiqué en conclusion du chapitre 4, l'un de nos objectifs à terme est de proposer un service de type client/serveur, d'aide à la décision pour la répartition dynamique de charge dans les réseaux de stations de travail. Toutefois, les perspectives offertes par un tel service ne se limitent pas au seul domaine de la répartition dynamique de charge. En effet, l'observation du comportement des machines et du réseau peut aussi s'avérer très utile afin de construire des outils de diagnostique et/ou de dimensionnement des installations.

En ce qui concerne le système MPCFS, nos objectifs sont, dans un premier temps, de diffuser le prototype auprès de la communauté Linux. En effet, nous souhaitons maintenant faire évoluer les spécifications du système, à partir des observations et évaluations ménées par les utilisateurs intéressés de cette communauté. Cette diffusion a aussi pour objectif de solliciter l'expertise des spécialistes et concepteurs de ce système pour optimiser les performances du prototype. Dans un deuxième temps, nous envisageons l'élaboration d'un standard auprès des communautés Internet et UNIX.

Parallèlement à ce travail de consolidation du prototype, nous aimerions aussi approfondir l'étude des protocoles de communication multipoints. Comme nous l'avons au chapitre 3, il semble difficilement envisageable de trouver un protocole qui réponde à l'ensemble des besoins en matière de communications multipoints. Nous aimerions donc évaluer et comparer les qualités et propriétés respectives des protocoles existants.

Cette comparaison est actuellement difficile, car la plupart de ces protocoles n'est disponible qu'en contexte utilisateur, et ce, au travers de bibliothèques ou d'environnements

spécifiques. Bien sûr, l'idéal pour établir cette comparaison serait de disposer d'implémentations de ces protocoles directement au niveau du système d'exploitation, au travers de l'API des *sockets*, par exemple. Mais cette adaptation est difficile à réaliser, en particulier pour des protocoles aussi complexes que XTP. Or, l'une des qualités intéressantes du système MPCFS réside justement dans sa capacité à pouvoir faire appel, au travers du mécanisme de VFS, à des fonctionnalités qui ne sont disponibles qu'en contexte utilisateur. MPCFS offre donc par ce biais la possibilité d'évaluer et de comparer, au travers d'une même API de programmation, et à l'aide de programmes de tests strictement identiques, les qualités et propriétés respectives des différents protocoles existants.

Annexe A

Guide utilisateur de *LoadBuilder*

A.1 Utilisation du client

A.1.1 Format des lignes de commande

L'interface utilisateur du client est un interpréteur de commandes simples, qui offre deux modes de fonctionnement légèrement différents : un mode prévu pour du traitement par lots, dans lequel les sorties produites se limitent au strict minimum nécessaire, et un mode moins austère, prévu pour une utilisation interactive (affichage systématique d'un prompt, formattage plus convivial des données affichées).

L'interpréteur reconnaît 11 types de commande, débutant chacune par l'un des 11 mot-clefs présentés dans la première colonne du tableau A.1. Chacun de ces mot-clefs peut, éventuellement, être suivi de paramètres, appartenant aux 5 catégories suivantes :

- **`site`** : désignation logique d'une machine (par exemple `www.inria.fr`) ;
- **`module`** : le nom d'un module de charge synthétique, d'un module de collecte de statistiques ou d'un module d'évaluation de performance ;
- **`global_id`** : un identifiant global de processus. Un identifiant global est formé du caractère # suivi du numéro de séquence de la commande ayant lancé le processus ;
- **`local_id`** : un numéro de processus UNIX (`pid`) ;
- **`param`** : des paramètres qui doivent être transmis au programme qui exécute un module, lors de son lancement. Ces paramètres peuvent être délimités par des guillemets pour éviter les ambiguïtés ;
- **`param_cmd`** : un paramètre de type entier ou chaîne de caractères (délimitée par des guillemets).

Pour être reconnues par l'interpréteur, les lignes de commandes doivent impérativement respecter l'une des 8 syntaxes suivantes :

1. **`mot-clef`**
2. **`mot-clef site1`** [**`site2`** ...]
3. **`mot-clef module1`** [**`module2`** ...]
4. **`mot-clef global_id1`** [**`global_id2`** ...]
5. **`mot-clef site module1`** [**`module2`** ...]
6. **`mot-clef site module1 param1`** [**`module2 param2`** ...]
7. **`mot-clef site id_processus1`** [**`id_processus2`** ...]
8. **`mot-clef param_cmd`**

Mot-clef	Syntaxe	Description
QUIT	1	Termine l'exécution du client
HELP	1,3	Aide en ligne
RINFO	1,2,3,4,5,7	Récupère les informations d'état concernant l'état des modules auprès des serveurs
RUN	5,6	Lance un module
START	5,6	Lance un module (identique RUN)
TERM	1,2,3,4,5,7	Termine l'exécution d'un module en conservant son état dans la mémoire du serveur
KILL	1,2,3,4,5,7	Termine l'exécution d'un module et supprime définitivement son état de la mémoire du serveur
GET	1,2,3,4,5,7	Récupère les données affichées par les modules sur leurs sortie standard et sortie d'erreur
WAIT	1,2,3,4,5,7,8	Bloque le client dans l'attente d'un délai ou de la terminaison de modules
COLL	1,2,3,4,5,7	Récupère les données collectées et sauvegardées par les modules (statistiques, mesures de performance)
ECHO	1,8	Répète son paramètre sur la sortie standard (surtout utile pour le mode de traitement par lot)

TAB. A.1: *Les commandes du client LoadBuilder*

A.1.2 Syntaxe spécifique de chaque commande

Commande QUIT. La commande QUIT interrompt la session et termine l'exécution du client. Cette commande ne prend pas de paramètres.

La session peut être reprise en relançant l'exécution du client. Cependant, le client ne sait pas retrouver automatiquement l'état d'une session interrompue. Pour lui permettre de reconstruire cet état, il faut lui demander d'interroger chacun des sites impliqués dans l'expérience à l'aide de la commande RINFO.

Commande HELP. La commande HELP sans paramètre affiche un écran d'aide général. La commande HELP suivi d'un nom de module affiche un écran d'aide spécifique au module.

Commande RINFO. La commande RINFO interroge les serveurs afin de récupérer les informations d'état concernant les modules ou sites indiqués en paramètres. Ces informations (en particulier le nom des machines sur lesquelles des serveurs sont actifs) sont conservées localement par le client.

Cette commande accepte les syntaxes de type 1,2,3,4,5 et 7 :

– syntaxe 1 : interroge l'ensemble des serveurs déjà connus par le client ;

Catégorie	Nom	Description
Charge synthétique	`LCPU`	Charge du processeur
	`LMEM`	Charge de la mémoire
	`LSYS`	Charge du système d'exploitation
	`LUSVR`	Générateur de trafic UDP/IP
	`LUCLNT`	Consommateur de trafic UDP/IP
	`LTSVR`	Générateur de trafic TCP/IP
	`LTCLNT`	Consommateur de trafic TCP/IP
Évaluation de performances	`MCPU`	Mesure de performance du processeur
	`MMEM`	Mesure de performance de la mémoire
	`MSYS`	Mesure de performance du système d'exploitation
	`UPING`	Mesure de performance UDP/IP (test "ping-pong")
	`UPONG`	
	`TPING`	Mesure de performance TCP/IP (test "ping-pong")
	`TPONG`	
Collecte de statistiques	`LSTAT`	Statistiques locales
	`NSTAT`	Statistiques réseau

TAB. A.2: *Les modules LoadBuilder*

– syntaxe 2 : interroge une liste explicite de serveurs. En début de session, cette commande permet en particulier de vérifier la présence des serveurs sur les machines participant à une expérience ;

– syntaxe 3 : interroge l'ensemble des serveurs déjà connus à propos des modules dont les types sont indiqués en paramètre. Chaque serveur retourne l'état des éventuels modules dont il a la charge qui correspondent à cette liste ;

– syntaxe 4 : récupère l'état des modules dont les numéros de séquence de lancement dans la session sont donnés en paramètre ;

– syntaxe 5 : interroge un (unique) serveur à propos des modules dont les types sont donnés en paramètres ;

– syntaxe 7 : interroge un site à propos des modules dont les numéros de processus UNIX sont donnés en paramètre.

Exemples :

```
rinfo                    # récupère les informations concernant chacun
```

```
                        # des modules sur chacune des machines connues
                        # du client
---------------------------------------------------------------------
rinfo lstat uping       # récupère les informations concernant chacun
                        # des modules de type lstat et uping qui
                        # s'exécutent sur les machines connues du client
---------------------------------------------------------------------
rinfo www               # récupère les informations concernant
                        # l'ensemble des modules qui s'exécutent sur la
                        # machine www
```

Les commandes `TERM, KILL, GET` et `COLL`, que nous décrivons dans les paragraphes suivants, acceptent la même syntaxe et traitent donc leurs arguments de façon similaire.

Commandes `RUN` et `START`. Ces deux commandes sont identiques. Elles permettent le lancement d'un ou plusieurs modules sur une machine. Si le site n'est pas encore connu par le client, ce dernier l'ajoute automatiquement à sa liste s'il détecte la présence d'un serveur sur ce site. Dans le cas contraire, la commande est ignorée et le client retourne une erreur (non fatale).

Exemples :

```
start www lstat nstat          # lance l'exécution des mo-
dules lstat
                               # et nstat sur la machine www
---------------------------------------------------------------------
start aspic uping "-o python"  # lance l'exécution du module
                               # uping sur la machine aspic, avec
                               # les paramètres "-o python"
```

Commande `TERM`. Cette commande termine l'exécution des modules indiqués en paramètres et retourne leur état. L'état retourné peut cependant ne pas refléter le résultat de la commande, car la terminaison d'un module peut exiger quelques instants, le temps que le processus sauvegarde les données qu'il conserve en mémoire (l'interruption est obtenue par l'envoi d'un signal masquable).

Commande `KILL`. Cette commande termine aussi l'exécution des modules indiqués en paramètre, mais à la différence de la commande précédente, l'état des modules interrompus est effacé de la mémoire des serveurs. De plus, l'interruption des processus est forcée, par l'envoi d'un signal non masquable.

Commande `GET`. Sur chaque machine, les serveurs interceptent et conservent les données affichées sur les sorties standard et d'erreur par les processus exécutant chaque module. La commande `GET` permet de récupérer ces affichages et de les reproduire sur les sorties équivalentes (standard et erreur) du client.

Commande `WAIT`. Lorsqu'elle est utilisée selon la syntaxe 8, la commande `WAIT` bloque l'exécution du client pour une durée correspondant au nombre de secondes indiquées en paramètres. Lorsqu'elle est utilisée selon les syntaxes 1,2,3,4,5et 7, cette bloque le client jusqu'à ce que l'ensemble des modules indiqués en paramètres aient terminé leur exécution.

Exemples :

```
wait 10              # attend 10 secondes
-------------------------------------------------------------------
wait                 # attend la terminaison de tous les
                     # modules sur toutes les machines connues du
                     # client
-------------------------------------------------------------------
wait uping upong     # attend la terminaison de l'ensemble des
                     # modules uping et upong s'exécutant sur les
                     # machines connues du client
-------------------------------------------------------------------
wait www             # attend la terminaison de l'ensemble des
                     # modules qui s'exécutent sur la machine www
-------------------------------------------------------------------
wait www tping       # attend la terminaison de l'ensemble des
                     # modules tping qui s'exécutent sur la machine
                     # www
```

Commande `COLL`. Cette commande permet de transférer les données sauvegardées par les modules de mesure et de collecte des statistiques. Ces données, qui sont sauvegardées par les modules sous forme binaire pour des raisons de compacité sont converties vers le format binaire du client (à l'aide d'un encodage XDR). En réception, le client les sauvegarde à son tour, mais dans un format ASCII.

Commande `ECHO`. Lorsqu'elle est utilisée selon la syntaxe 1, la commande `ECHO` demande au client d'afficher un saut de ligne sur sa sortie standard. Lorsqu'elle est utilisée selon la syntaxe 8, cette commande demande au client d'afficher la chaîne de caractères qu'elle reçoit en paramètres. Cette commande est surtout utile afin d'insérer des commentaires et des points de repère dans les sorties produites par le client lorsqu'il est utilisé pour du traitement par lot.

A.2 Configuration de l'environnement *LoadBuilder*

Définition des variables d'environnement. Tous les programmes de l'environnement *LoadBuilder* partagent le même mécanisme de configuration, inspiré du mécanisme de configuration des ressources de X11. Le principe de ce mécanisme est de permettre la définition d'un nombre quelconque de variables d'environnement, qui définissent les options et paramètres de fonctionnement de chacun des programmes de *LoadBuilder*. Les noms de variables sont construits de façon hiérarchique : au niveau le plus bas, une variable

définit une option par défaut pour l'ensemble des programmes. Au niveau inférieur, chaque variable peut être spécialisée pour un exécutable ou une fonction particulière, par l'ajout de préfixes ou de suffixes au nom initial de la variable. Par exemple, la variable `DataDir` définit le répertoire par défaut dans lequel les données produites doivent être stockées. Comme l'exécutable du serveur s'appelle `LBd`, en définissant une variable `LBd.DataDir`, on substitue à la valeur par défaut précédente, une valeur spécifique pour l'exécutable `LBd`, c'est-à-dire pour le programme du serveur.

Ces variables d'environnement sont initialisées lors du lancement du programme. Par défaut, ces variables ont une valeur initialisée de façon statique. Cette valeur par défaut peut être redéfinie au travers d'un fichier de configuration ou directement au niveau des paramètres reçus en ligne de commande par le programme. Les définitions reçues en ligne de commande écrasent les éventuelles définitions lues dans un fichier de configuration, qui écrasent elles-mêmes les définitions statiques.

Il existe donc plusieurs façons de définir une même option ou un même paramètre de fonctionnement. Par exemple, dans le cas de la variable définissant le répertoire dans lequel les données produites par le serveur doivent être sauvegardées, nous avons les six possibilités suivantes (données dans l'ordre décroissant de leur priorité d'application) :

1. Donner une valeur à la variable `LBd.DataDir` en paramètre de la commande lançant l'exécution du serveur :

   ```
   $ LBd --LBd.DataDir ./Data/LBd/
   ```

2. Définir la variable `LBd.DataDir` dans le fichier de configuration du serveur, par une ligne telle que la suivante :

   ```
   LBd.DataDir=./Data/LBd/
   ```

3. Définir de façon statique la variable `LBd.DataDir` dans le programme `LBd` (notons que dans ce cas, les trois méthodes de définition qui suivent ne peuvent jamais être appliquées) ;

4. Définir la variable globale `DataDir` en paramètre de la commande lançant l'exécution du serveur ;

5. Définir la variable `DataDir` dans le fichier de configuration du serveur ;

6. Définir de façon statique la variable `DataDir` dans le programme (cette variable est actuellement initialisée avec le répertoire courant).

Variables d'environnement reconnues par le client et le serveur. Le tableau A.3 donne la liste des variables d'environnement reconnues par le client et le serveur.

Variable	Rôle
DataDir	Indique le répertoire dans lequel les données doivent être placées
Log.Pid	Insertion du numéro de processus dans chaque trace
Log.Perror	Duplique les traces sur la sortie d'erreur
Log.Level	Niveau de "bavardage" (similaire à la fonction syslog(3)). 0 indique le niveau le plus bas et 7 le niveau le plus haut.
Log.Syslog	Utiliser le service UNIX d'affichage de traces (syslog)
Log.Ident	Chaîne de carcatères à insérer dans chaque trace produite
Log.HostName	Insertion du nom de la machine dans chaque trace produite
Log.File	Nom du fichier de trace (placé dans le repertoire défini par DataDir)

TAB. A.3: *Variables d'environnement actuellement reconnues dans LoadBuilder*

Fichier de configuration. Par défaut le fichier de configuration utilisé par un programme *LoadBuilder* de nom `<prog>` s'appelle `.<prog>rc` et se trouve dans le répertoire racine de l'utilisateur qui lance le programme. Toutefois, il est possible de forcer le programme à utiliser un autre fichier de configuration au travers d'un paramètre donné en ligne de commande (option `-f`).

Les règles de définition des fichiers de configuration sont les suivantes :

– les lignes vides sont ignorées ;

– des commentaires peuvent être introduits. Ils sont signalés par le caractère `#`, qui doit obligatoirement apparaitre en début de ligne ;

– une définition de variable est une ligne non vide non commentée qui contient le caractère '`=`'. Tout ce qui précède le premier caractère `=` (espaces compris) est considéré comme étant le nom de la variable, et tout ce qui suit ce caractère est considéré comme son contenu. Quand aucun caractère ne suit le premier caractère `=`, la variable est initilisaée par défaut à 0 (zéro) ;

Exemple de fichier de configuration La figure A.1 présente un exemple de fichier de configuration.

```
##########################################################################
##          Exemple de fichier de configuration LoadBuilder
##########################################################################

##
## Définition globales (valeurs par défaut)
##


DataDir=.
## Insertion du numéro de processus dans chaque trace
Log.Pid=1
## Duplique les traces sur la sortie d'erreur
Log.Perror=0
## Niveau de "bavardage" (similaire à la fonction syslog(3))
## -1 ou indéfini=le plus bas, 7=le plus haut
Log.Level=7
## Utiliser le service UNIX d'affichage de traces (syslog)
Log.Syslog=0
## Chaîne de carcatères à insérer dans chaque trace produite
Log.Ident=LB(??)
## Insertion du nom de la machine dans chaque trace produite
Log.HostName=0
## Nom du fichier de trace (placé dans le repertoire dé-
fini par DataDir)
Log.File=LBlog

##
## Redéfinition spécifiques au client et au serveur
##

## Client
LB.Log.Ident=LB(client)
LB.Log.HostName=0

## Serveur
LBd.Log.Ident=LB(daemon)
LBd.Log.Perror=1
LBd.Log.File=LBdlog
LBd.Log.File.HostName=1
```

FIG. A.1: *Exemple de fichier de configuration LoadBuilder*

Annexe B

Configuration du système de fichier MPCFS

Nous décrivons dans cette annexe la syntaxe des deux principaux fichiers de configuration du système MPCFS : le fichier d'accès, qui définit les règles d'accès aux groupes de communication par les processus, et le fichier d'export, qui définit les règles d'association de machines dans les groupes de communication.

B.1 Fichier d'accès

Comme nous l'avons vu au paragraphe 5.3.3.2 (page 126), chaque ligne de ce fichier est formée de cinq champs, séparés par des espaces :

1. La désignation d'un ensemble de noms de groupes (pouvant être défini au moyen d'expressions régulières) ;
2. La désignation d'un ensemble d'utilisateurs ;
3. L'utilisateur et le groupe d'utilisateurs propriétaires du groupe lors de sa création ;
4. La liste des permissions et propriétés par défaut des fichiers et répertoires du groupe ;
5. La liste des modifications que l'utilisateur peut apporter à la configuration par défaut.

Ensemble de noms de groupes de communication. Cet ensemble est décrit par une liste formée d'expressions régulières. Cette liste est délimitée par des parenthèses et ses éléments sont séparés par des virgules. Les expressions régulières actuellement reconnues s'inspirent de celles qui sont couramment utilisées par les interpréteurs de commandes pour désigner des fichiers. En particulier la signification des méta caractères `*`, `?`, `[`, `]`, et \ est conservée :

- le méta caractère '`?`' désigne un caractère quelconque ;
- le méta caractère '`*`' désigne un suite quelconque de caractères ;
- un ensemble de caractères entre `[ ]` désigne un caractère parmi cet ensemble de caractères ;
- le méta caractère \ permet de "déspécialiser" le caractère qui le suit ;

De plus, le méta caractère '`$`' permet de faire référence à un certain nombre de variables dont les valeurs dépendent du contexte de l'évaluation :

- les variables `$u` et `$U` désignent respectivement l'`uid` et le nom de `login` de l'utilisateur pour qui l'évaluation est exécutée ;
- les variables `$g` et `$G` désignent respectivement le `gid` et le nom de groupe de l'utilisateur pour qui l'évaluation est exécutée ;

Exemples :

`(*)`	: tous les groupes
`(foo,bar,baz)`	: les groupes `foo, bar` et `baz`
`(grp_[a-z])`	: les groupes `grp_a, grp_b` ... `grp_z`
`(*_$u,*_$U)`	: les groupes dont le nom se termine par le nom ou l'identifiant UNIX de l'utilisateur

Ensemble d'utilisateurs. Cet ensemble est une liste parenthésée dont les éléments, séparés par des virgules, désignent soit des noms (*login*) d'utilisateurs, soit des noms de groupes d'utilisateurs.

Pour éviter tout conflit entre nom de groupe (d'utilisateurs) et nom d'utilisateur, ces derniers doivent être préfixés par le caractère ':'.

Le caractère spécial '*', utilisé seul, permet de désigner l'ensemble des utilisateurs reconnus sur la machine.

A l'inverse, une liste vide désigne un ensemble vide d'utilisateurs (ce qui permet d'interdire à qui que ce soit l'utilisation d'un groupe).

Exemples :

`(*)`	: l'ensemble de tous les utilisateurs
`()`	: aucun utilisateur
`(sloop,u-mpcfs)`	: le groupe `sloop` et le groupe `u-mpcfs`
`(:od,:ms,admin)`	: les utilisateurs `od, ms` et le groupe `admin`

Utilisateur et le groupe d'utilisateurs propriétaires. L'utilisateur et le groupe d'utilisateurs propriétaires du groupe de communication sont présentés sous la forme `user.group`. S'il s'agit d'un utilisateur différent de celui du propriétaire effectif du processus qui réalise la demande, les éventuels paramètres de la demande concernant les permissions sur le répertoire du groupe, le fichier `activate` et les fichiers contenant les messages sont refusés. La variable `$u` permet de désigner le propriétaire effectif du processus qui fait la demande, et la variable `$g` son groupe effectif.

Liste des permissions et propriétés par défaut. La liste des permissions et propriétés par défaut des fichiers et répertoires du groupe est comoposée des élements suivants (ou `xxx` désigne un nombre en octal) :

- `g:xxx` : permissions du répertoire du groupe ;
- `s:xxx` : permissions des répertoires de multiplexage des processus membre ;
- `a:xxx` : permissions du fichier d'association ;
- `n:xxx` : permissions des nœuds de communication ;
- `m:bbb` : permissions des fichiers de message (`b` = `0` ou `1`) ;
- `+wb` ou `-wb` : indiquent si les écritures doivent ou non être bloquantes par défaut ;
- `+rb` ou `-rb` : indiquent si les lectures doivent être bloquantes ou non par défaut.

`gsan:xxx` est une notation abrégée pour `g:xxx,s:xxx,a:xxx,n:xxx` ; `A:xxx` est équivalent à `gsan:xxx` ; `+` est équivalent à `+wb,+rb`, et `-` est équivalent à `-wb,-rb`.

La liste des modifications que l'utilisateur peut apporter à la configuration par défaut, au travers des paramètres d'enregistrement, adopte une syntaxe identique à celle du champs précédent, à ceci près que le nombre octal est un masque indiquant si chacun des bits de permission peut être modifié. Le mode bloquant ou non bloquant n'est pas affecté par ce champ.

Exemple de configuration de groupes.

```
#
# Exemple de /etc/mpc_groups
#

# private_<login>_* est reservé à l'utilisateur <login>, et son
# accès restreint
(private_$U_*) (*) $u.$g (A:700,m:100,+wb,+rb) (A:777,m111)

# Tout autre groupe commencant par 'private_' est interdit
(private_*) () root.root

# Le groupe info appartient à root, quel quel soit le processus
# qui demande sa création. Son accès est libre, mais l'écri-
ture y est
# interdite. La création de sous-groupes et les associations de
# machines y sont en revanche autorisées.
(info) (*) root.root (g:755,as:777,n:111,m:111,+) (A:000,m:000)

# Par défaut, les groupes appartiennent aux processus qui en font
# la demande
(*) (*) $u.$g (A:775,m:110,+) (A:777,m111)
```

B.2 Fichier d'export

Le fichier d'export définit les permissions d'association d'un groupe de la machine locale avec les groupes existants sur les autres machines. Chaque ligne significative de ce fichier est formée :

- d'un champ désignant un ensemble de groupes (pouvant être défini au moyen d'expressions régulières) ;

- une succession de champs désignant chacun :

  - une machine ou un ensemble de machines (domaine) ;
  - le type d'interaction autorisée vis-à-vis de cette(ces) machine(s).

Désignation des groupes. Les groupes de communication sont désignés par une liste formée d'expressions régulières. Les expressions régulières reconnues sont similaires à celles que nous utilisons la désignation des groupe dans le fichier d'accès. Cependant, les variables $u et $U $g et $G ont une signification différente. Elles représentent l'utilisateur et le groupe d'utilisateurs du propriétaire du fichier d'association (`activate`) à partir duquel la demande est faite.

Désignation des machines. Chacun des champs de la partie désignant les machines est formé de deux parties.

La première partie désigne une machine ou un ensemble de machines. Elle est formée à partir d'expressions régulières simplifiées, dans lesquelles seul le meta-caractère `*` est reconnu.

La deuxième partie est optionnelle. Elle définit le type d'interaction autorisée entre la machine locale et l'ensemble des machines désignées par la première partie, pour tout groupe de communications désigné par le premier champ de la ligne. Cette partie contient l'un des drapeaux suivant, placé entre parenthèses :

- `-r` les messages en provenance de ces machines pour ces groupes de communications sont systématiquement rejetés ;
- `-w` aucun des messages transmis par un processus local dans ces groupes de communications n'est envoyé à destination de ces machines ;
- `-rw` ces machines ne peuvent en aucune façon être associées à ces groupes ;

Par défaut, les machines peuvent être associées à ces groupes sans restriction.
Exemple de fichier d'export :

```
(upriv*,gpriv*) *.inria.fr(r-)
(IRClocal) *.inria.fr() *(w-)
(IRCfr) *.fr() *(w-)
(*local*) *.inria.fr()
(*) *()
```

Bibliographie

[Anderson 94] Scott D. Anderson, Adam Carlson, David L. Westbrook, David M. Hart et Paul R. Cohen. *Tools for experiments in planning*. Proceedings of the International Conference on Tools with Artificial Intelligence, pages 615–623, 1994.

[André 88] Françoise André et Jean-Louis Pazat. *Le placement de tâches sur des architectures parallèles*. Techniques et Sciences Informatiques, vol. 7, no. 4, 1988.

[Armstrong 92] S. Armstrong, A. Freier et K. Marzullo. *RFC 1301: Multicast Transport Protocol*, Février 1992. Status: INFORMATIONAL.

[Arpaci 95] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson et David A. Patterson. *The interaction of parallel and sequential workloads on a network of workstations*. In Proceedings of the Joint International Conference on Measurement & Modeling of Computing Systems, Ottawa, Canada, 1995.

[Bach 86] Maurice J. Bach. The Design of the UNIX Operating System. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1986.

[Barak 85] Amnon Barak et Amnon Shiloh. *A Distributed Load-balancing Policy for a Multicomputer*. Software – Practice and Experience, vol. 15, no. 9, pages 901–913, Septembre 1985.

[Barak 98] Amnon Barak et Oren La'adan. *The MOSIX Multicomputer Operating System for High Performance Cluster Computing*. Journal of Future Generation Computer Systems, vol. 13, no. 4–5, pages 361–372, 1998.

[Beck 98] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus et Dirk Verworner. Linux Kernel Internals. Addison-Wesley, 2 edition, 1998.

[Bernard 91a] Guy Bernard et Michel Simatic. *A Decentralized and Efficient Algorithm for Load Sharing in Networks of Workstations*. In

Proc. EurOpen Spring '91 Conference, Tromso (Norway), Mai 1991.

[Bernard 91b] Guy Bernard, D. Stève et Michel Simatic. *Placement et migration de processus dans les systèmes répartis faiblement couplés*. Techniques et Sciences Informatiques, vol. 10, no. 5, Mai 1991.

[Bernard 96] Guy Bernard et Bertil Folliot. *Caractéristiques Générales du Placement Dynamique : Synthèse et Problématique*. In Actes de l'école d'été «Placement dynamique et répartition de charge : application aux systèmes parallèles et répartis», Presqu'île de Giens, France, Juillet 1996. INRIA.

[Bernaschi 95] M. Bernaschi et G. Richelli. *PVMe: an enhanced implementation of PVM for the IBM 9076 SP2*. Lecture Notes in Computer Science, vol. 919, pages 461–471, 1995.

[Bernon 95] Carole Bernon. *Conception et évalution d'une plate-forme pour le placement dynamique de processus communiquants*. Thèse de doctorat, IRIT, Septembre 1995.

[Berry 91] Robert Berry et Joseph Hellerstein. *An Approach to Detecting Changes in the Performance Characteristics of Computer Systems*. Performance Evaluation Review : ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, vol. 19, no. 1, Mai 1991.

[Bertsekas 89] Dimitri P. Bertsekas et John N. Tsitsiklis. Parallel and Distributed Computation. Prentice-Hall Inc., New Jersey, 1989.

[Birman 85] Kenneth P. Birman, Amr El Abbadi, Wally Dietrich, Thomas A. Joseph et Thomas Raeuchle. *An Overview of the ISIS Project*. IEEE Distributed Processing Technical Committee Newsletter, Janvier 1985.

[Birman 86] Kenneth P. Birman et Thomas A. Joseph. *Communication Support for Reliable Distributed Computing*. In Proc. Asilomar Workshop on Fault Tolerant Distributed Computing, 1986. Also available as TR86-753 from Cornell University. Basis for RFC 992.

[Birman 94] Kenneth P. Birman et Robbert van Renesse, editeurs. Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, 1994.

[Black 95] Uyless Black. ATM: Foundation for Broadband Network. Series in advanced communications technologies. Prentice Hall, 1995.

[Boden 95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic et Wen-King Su. *Myrinet: A Gigabit-per-Second Local Area Network*. IEEE Micro, Février 1995.

[Boutaba 92] Raouf Boutaba et Bertil Folliot. *Load Balancing in Local Area Networks*. In Proceedings of the Networks'92 Intl. Conf. on Computer Networks, Architecture and Applications, pages 73–89, Trivandrum, India, Octobre 1992. IFIP & TC6.

[Bouvry 93] Pascal. Bouvry, Jean-Marc Geib et Denis Trystram. *Répartition de charge*. In École d'automne CAPA'93 : conception et analyse d'algorithmes parallèles, pages 167–194, Port d'Albret, Landes, Septembre 1993. Action CAPA du PRC C^3.

[Braudes 93] R. Braudes et S. Zabele. *RFC 1458: Requirements for Multicast Protocols*, Mai 1993. Status: INFORMATIONAL.

[Burgevin 89] Patrice Burgevin, André Couvert et René Pedronon. *Délai de communication entre noeuds voisins sur l'iPSC/2*. Rapport technique RR-467, Institut de Recherche en Informatique et Systèmes Alèatoires (IRISA), 1989.

[Burns 94] Greg Burns, Raja Daoud et James Vaigl. LAM: An open cluster environment for MPI. 1994.

[Cabillic 96] Gilbert Cabillic et Isabelle Puaut. *Stardust: An Environment for Parallel Programming on Networks of Heterogeneous Workstations*. In Proc. of the Second Int'l Euro-Par Conf., volume I, pages 114–119, Août 1996.

[Cabrera 86] Luis-Felipe Cabrera. *The influence of workload on load balancing strategies*. In Proceedings of the USENIX Summer Conference, Atlanta, June 1986.

[Callaghan 95] B. Callaghan, B. Pawlowski et P. Staubach. *RFC 1813: NFS Version 3 Protocol Specification*, Juin 1995. See also RFC1094. Status: INFORMATIONAL.

[Calzarossa 93] Maria Calzarossa et Giuseppe Serazzi. *Workload Characterization: A survey*. Proceedings of the IEEE, vol. 81, no. 8, Aug 1993.

[Calzarossa 95] M. Calzarossa, G. Haring, G. Kotsis, A. Merlo et D. Tessera. *A hierarchical approach to workload characterization for parallel systems*. In B. HertzBerger et G. Serazzi, editeurs, LNCS(919). HPCN EUROPE, Springer Verlag, 1995.

[Cap 93] Clemens H. Cap et Volker Strumpen. *Efficient parallel computing in distributed workstation environnements*. Parallel Computing, vol. 19, pages 1221–1234, 1993.

[Cappello 98] Franck Cappello et Olivier Richard. *Architectures parallèles à partir de réseaux de stations de travail : réalités, opportunités, enjeux*. Calculateurs Parallèles, vol. 10, no. 1, 1998.

[Card 93] Remy Card. *MASIX un Système d'Exploitation Multi-Environnements utilisant le Micro-Noyau MACH*. Thèse de doctorat, Université de Paris 6, 1993.

[Casavant 88] Thomas L. Casavant et Jon G. Kuhl. *A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems*. IEEE Transactions on Software Engineering, vol. 14, no. 2, pages 141–154, Février 1988.

[Chatonnay 98] Pascal Chatonnay. *Gestion de l'allocation des ressources aux objets dans les systèmes répartis, une approche multicritère intégrant les communications*. Thèse de doctorat, Université de Franche-Comté, mon 1998.

[Chou 82] Timothy C. K. Chou et Jacob A. Abraham. *Load Balancing in Distributed Systems*. IEEE Transactions on Software Engineering, vol. SE-8, no. 4, Juillet 1982.

[Coffman 76] E. G. Coffman. Computer and job-shop scheduling theory. John Wiley, New York, 1976.

[Comer 93] Douglas E. Comer et David L. Stevens. Client-server programming and applications (BSD socket version), volume III of *Internetworking with TCP/IP*. Prentice-Hall International Editions, 1993.

[Cosquer 94] François J. N. Cosquer et Paulo Verissimo. *Survey of Selected Groupware Applications and Supporting Platforms*. Technical Report BROADCAST#TR94-40, ESPRIT Basic Research Project BROADCAST, Octobre 1994.

[Custer 93] Helen Custer. Inside windows NT. Microsoft Press, 1993.

[Dabbous 93] Walid Dabbous et C. Huitema. *XTP implementation under Unix*. Technical Report RR-2102, Inria, Institut National de Recherche en Informatique et en Automatique, Novembre 1993.

[Dalle 93] Olivier Dalle. Mesure des performances de communication du multiprocesseur meiko. Mémoire de DEA, Réseaux et Systèmes Distribués (RSD), Université de Nice – Sophia Antipolis, 1993.

[Dalle 96] Olivier Dalle. *LoadBuilder: a tool for generating and modeling workloads in distributed workstations environements*. In K. Yetongnon et S. Hariri, editeurs, Proceedings of the 9th International Conference on Parallel & D istributed Computing Systems. ISCA, 1996.

[Dalle 98a] Olivier Dalle. *MPCFS : un exemple d'intégration transparente de me'canismes de communication multipoints dans les systèmes UNIX*. In Dominique Méry et Guy-René Perrin, editeurs, 10e Rencontres Francophones du Parallélisme (RENPAR'10), Strasbourg, Juin 1998.

[Dalle 98b] Olivier Dalle. *MPCFS : un système de fichiers virtuel pour communications multipoints fiables entre systèmes UNIX*. In 2e Journées Doctorales Informatique et Réseaux (JDIR'98), Paris, Avril 1998.

[Deering 89] S. E. Deering. *RFC 1112: Host extensions for IP multicasting*, Août 1989. Obsoletes RFC0988, RFC1054. Updated by RFC2236. Status: STANDARD.

[Desprez 90] Frédéric Desprez et Bernard Tourancheau. *Modélisation des performances de communication sur le tnode avec le Logical system transputer toolset*. La lettre du transputer et des calculateurs distribués, vol. Septembre, pages 65–77, Septembre 1990.

[Devarakonda 89] Murthy V. Devarakonda et Ravishankar K. Iyer. *Predictability of Process Resource Usage: A Measurement-Based Study on UNIX*. IEEE Transactions on Software Engineering, vol. 15, no. 12, pages 1579–86, Décembre 1989.

[Dolev 96] Danny Dolev et Dalia Malki. *The Transis Approach to High Availability Cluster Communication*. Communications of the ACM, vol. 39, no. 4, Avril 1996.

[Douglis 91] F. Douglis et J. Ousterhout. *Transparent Process Migration: Design Alternatives and the Sprite Implementation*. Software Practice and Experience, vol. 21, no. 9, pages 757–785, Novembre 1991.

[Efe 89] K. Efe et B. Groselj. *Minimizing Control Overheads in Adaptive Load Sharing*. In Proceedings of the 9th Int. Conf. on Distributed Computing Systems, pages 307–315, Newport Beach, Juin 1989. IEEE.

[Faulkner 91] Roger Faulkner et Ron Gomes. *The Process File System and Process Model in UNIX System V*. In USENIX Association, editeur, Proceedings of the Winter 1991 USENIX Conference: January 21-January 25, 1991, Dallas, TX, USA, pages 243–252, Berkeley, CA, USA, Janvier 1991. USENIX.

[Ferrari 86] Domenico Ferrari et Songnian Zhou. *A load index for dynamic load balancing*. In Proceedings of the Fall Joint Computer Conference, pages 684–690, November 1986.

[Ferrari 88] Domenico Ferrari et Songnian Zhou. *An empirical investigation of load indices for load balancing applications*. In P.-J. Courtois et G. Latouche, editeurs, Proceedings of 12th IFIP WG 7.3 International Symposium on Computer Performance PERFORMANCE'87, pages 515–528, Brussels, Belgium, Décembre 1988. Elsevier Science Publishers (North-Holland).

[Folliot 92] Bertil Folliot. *Méthodes et outils de partage de charge pour la conception et la mise en œuvre d'applications dans les systèmes répartis hétérogènes*. Thèse de doctorat, Université de Paris 6, 1992.

[Fonlupt 95] Cyril Fonlupt. *Equilibrage de Charge pour Machines Parallèles Synchrones. Modèle, Formalisme et Analyse.* In Journées de Recherche sur le Placement Dynamique et la Répartition de Charge : Application aux Systèmes Répartis et Parallèles, Université Pierre et Marie Curie - Paris, 1995.

[Geist 94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek et Vaidy Sunderam. PVM - Parallel Virtual Machine : a user's guide and tutorial for networked parallel computing. MIT press, 1994.

[Gonzalez, Jr. 77] Mario J. Gonzalez, Jr. *Deterministic Processor Scheduling*. ACM Computing Surveys, vol. 9, no. 3, pages 173–204, Septembre 1977.

[Graham 95] Johm R. Graham. *Inside Virtual File Systems*. SunExpert Magazine, vol. 6, no. 9, pages 64–71, Septembre 1995.

[Gropp 96] William Gropp, Ewing Lusk, Nathan Doss et Anthony Skjellum. *High-performance, portable implementation of the*

MPI Message Passing Interface Standard. Parallel Computing, vol. 22, no. 6, pages 789–828, Septembre 1996.

[Gustavson 92] David B. Gustavson. *The Scalable Coherent Interface and related standards projects*. IEEE Micro, vol. 12, no. 1, pages 10–22, Février 1992.

[Guyennet 93] Hervé Guyennet et François Spies. *Étude Comparative de différents algorithmes de répartition de charges dans les systèmes distribués*. La Lettre du Transputer, pages 31–55, Juin 1993.

[Hall 94] Kara Ann Hall. The implementation and evaluation of reliable ip multicast. Master's thesis, University of Tennessee, Knoxville, USA, 1994.

[Harchol-Balter 96] Mor Harchol-Balter et Allen Downey. *Exploiting Process Lifetime Distributions for Dynamic Load Balancing*. In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer SYstems, volume 24,1 of *ACM SIGMETRICS Performance Evaluation Review*, pages 13–24, New York, Mai23–26 1996. ACM Press.

[Hémery 94] Fred Hémery. *Étude de la répartition dynamique d'activités sur architectures décentralisées*. Thèse de doctorat, Université des sciences et technologies de Lille, 1994.

[Huitema 95] Christian Huitema. Le routage dans l'Internet. Eyrolles, 1995.

[Huitema 96] Christian Huitema. IPv6 : The new internet protocol. Prentice Hall, 1996.

[Hwang 82] Kai Hwang, William J. Croft, George H. Goble, Benjamin W. Wah, Faye A. Briggs, Williams R. Simmons et Clarence L. Coates. *A Unix-Based Local Computer Network with Load Balancing*. IEEE Computer, vol. 15, Avril 1982.

[Jacobson 88] V. Jacobson. *Congestion Avoidance and Control*. ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988, vol. 18, 4, pages 314–329, 1988.

[Jacqmot 89] C. Jacqmot, E. Milgrom, W. Joossen et Y. Berbers. *Unix and load-balancing : a survey*. In EUUG'89, pages 1–15, April 1989.

[Jain 91] Raj Jain. The art of computer performance analysis : techniques for experimental design, measurement, simulation and modeling. Wiley, 1991.

[Jia 96] X. Jia, J. Cao, W. Jia et C.H. Lee. *A Discussion of Distributed System Environments and Distributed Operating Systems*. In K. Yetongnon et S. Hariri, editeurs, Proceedings of the 9th International Conference on Parallel & Distributed Computing Systems. ISCA, 1996.

[Johnson 96] Howard W. Johnson. Fast Ethernet: dawn of a new network. Prentice Hall, 1996.

[Kaashoek 92] Frans M. Kaashoek, Andrew S. Tanenbaum et Kees Verstoep. *Group Communication in Amoeba and its Applications*. In OpenForum, pages 365–381, Utrecht (the Netherlands), Novembre 1992.

[Kazar 90] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Ben A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Antony Mason, Shu-Tsui Tu et Edward R. Zayas. *DEcorum File System Architectural Overview*. In USENIX Association, editeur, Proceedings of the Summer 1990 USENIX Conference: June 11–15, 1990, Anaheim, California, USA, pages 151–164, Berkeley, CA, USA, Summer 1990. USENIX.

[Kleiman 86] Steve R. Kleiman. *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*. In USENIX Summer Conference, Atlanta, Juin 1986.

[Koch 98] Povl T. Koch et Xavier Rousset de Pina. *Flexible Operating System Support for SCI Clusters*. In Proceedings EuroPar'98, Southampton, Septembre 1998. À paraître.

[Koifman 96] Alex Koifman et Stephen Zabele. *RAMP: A Reliable Adaptive Multicast Protocol*. In IEEE INFOCOM'96, San Francisco, CA, USA, Mars 1996. Available from `http://www.tascnets.com/tbone/ramp.html`.

[Kunz 91] T. Kunz. *The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme*. IEEE Transactions on Software Engineering, vol. 17, no. 7, pages 725–730, Juillet 1991.

[Leffler 89] S. J. Leffler, M. K. McKusick, M. J. Karels et J. S. Quarterman. The design and implementation of the 4.3 BSD UNIX operating system. Addison-Wesley Publishing Company, 1989.

[Leland 86] Will E. Leland et Teunis J. Ott. *Load-balancing heuristics and process behavior*. In Performance'86 and ACM SIGMETRICS 1986, Vol 14. ACM Performance Evaluation Review, May 1986.

[Levy 90] Eliezer Levy et Abraham Silberschatz. *Distributed File Systems: Concepts and Examples*. ACM Computing Surveys, vol. 22, no. 4, pages 321–374, Décembre 1990.

[Liang 90] Luping Liang, Samuel T. Chanson et Gerald W. Neufeld. *Process Groups and Group Communications: Classifications and Requirements*. Computer, vol. 23, no. 2, pages 56–66, Février 1990.

[Litzkow 88] Michael J. Litzkow, Miron Livny et Matt W. Mutka. *Condor – A Hunter of Idle Workstations*. In Proc. of the 8th Int. Conf. on Distributed Computing Systems, pages 104–111, San Jose, CA, Juin 1988.

[Malki 94] Dalia Malki. *Multicast Communications for High Availability*. Ph.d. thesis, Hebrew University, Jerusalem, Israël, 1994. Available from `ftp://ftp.cs.huji.ac.il/-users/transis/thesis/`.

[Mankin 98] A. Mankin, A. Romanow, S. Bradner et V. Paxson. *RFC 2357: IETF Criteria for Evaluating Reliable Multicast Transport and Application Protocols*. IETF Transport Area working group, Juin 1998. Status: INFORMATIONAL.

[McCanne 93] Steven McCanne et Van Jacobson. *The BSD Packet Filter: A New Architecture for User-level Packet Capture*. In USENIX Association, editeur, Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA, pages 259–269, Berkeley, CA, USA, Janvier 1993. USENIX.

[Mehra 93] Pankaj Mehra. *Automated learning of load-balancing strategies for a distributed computer system*. Ph.d. thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1993.

[Melliar-Smith 90] Peter M. Melliar-Smith, Louise E. Moser et Vivek Agrawala. *Broadcast protocols for distributed systems*. IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1, pages 17–25, Janvier 1990.

[Melliar-Smith 91] Peter M. Melliar-Smith et Louise E. Moser. *Performance Analysis of a Broadcast Communications Protocol*. Performance Evaluation Review, vol. 20, pages 1–10, 1991.

[Mentat 94] Mentat. *Mentat XTP for STREAMS*, 1994. `http://-www.mentat.com/xtp/xtp.html`.

[Mirchandaney 89] R. Mirchandaney, D. Towsley et J.A. Stankovic. *Analysis of the Effects of Delays on Load Sharing*. IEEE Transactions on Computers, vol. 38, no. 11, pages 1513–1525, Novembre 1989.

[Moser 95] L. E. Moser, Peter M. Melliar-Smith, D. A. Agarwal, R. K. Budhia et C. A. Lingley-Papadopoulos. *Totem: A Fault-Tolerant Multicast Group Communication System*. In Proceedings of the 25th IEEE Int'l Conf. on Fault-Tolerant Computing, Passadena, CA, Juin 1995.

[MPI Forum 94] MPI Forum. MPI : A message passing interface. Draft, 1994.

[MPI Forum 96] MPI Forum. *2nd European MPI Workshop*. In MPI-2: Extensions to the message-passing interface, Vienna, Jan 1996.

[Mullender 90] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse et H. van Staveren. *Amoeba: A distributed operating system for the 1990s*. IEEE Computer, vol. 23, no. 5, pages 44–53, Mai 1990.

[Padovano 93] Michael Padovano. Networking applications on UNIX system V release 4. Prentice Hall, 1993.

[Partridge 93] Craig Partridge. Gigabit networking. Professional Computing Series. Addison-Wesley, 1993.

[Paté 98] David Paté, Stella Marc-Zwecker et Jean-Jacques Pansiot. *La diffusion de vidéo hiérarchique sur Internet*. In Deuxième Journées Doctorales Informatique et Réseaux (JDIR), Paris, Avril 1998.

[Paul 97] S. Paul, K. K. Sabnani, J. C. Lin et S. Bhattacharyya. *Reliable Multicast Transport Protocol (RMTP)*. IEEE Journal on Selected Areas in Communications, vol. 15, no. 3, pages 407–421, Avril 1997.

[POSIX.1 90] System application program interface (API) [C language]. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1990. Standard `ISO/IEC 9945-1: 1990`; `IEEE Std 1003.1-1990`.

[Pozzetti 95] E. Pozzetti, V. Vetland, J. Rolia et G. Serazzi. *Characterizing the resource demands of TCP/IP*. In B. HertzBerger et G. Serazzi, editeurs, LNCS(919). HPCN EUROPE, Springer Verlag, 1995.

[Prylli 98] L. Prylli et B. Tourancheau. *BIP: A New Protocol Designed for High Performance Networking on Myrinet*. Lecture Notes in Computer Science, vol. 1388, pages 472–??, 1998.

[Rao 96] Vandana M. Rao. Performance enhancement of the Totem system using the STREAMS mechanism. Master's thesis, University of California, Janvier 1996.

[Richard 98] Olivier Richard et Franck Cappello. *Sur la nature auto-similaire de l'activité de stations de travail et de serveurs HTTP*. Technique et Science Informatiques (TSI), vol. 17, no. 5, Mai 1998.

[Rifkin 86] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah et Kang Yueh. *RFS Architectural Overview*. In USENIX Association, editeur, Summer conference proceedings, Atlanta 1986: June 9–13, 1986, Atlanta, Georgia, USA, pages 248–259, P.O. Box 7, El Cerrito 94530, CA, USA, Summer 1986. USENIX.

[Rose 90] Marshall T. Rose. The Open Book: A Practical Perspective on Open Systems Interconnection. Prentice-hall, 1990. ISBN 0–13–643016–3.

[Rozier 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard et W. Neuhauser. *CHORUS Distributed Operating System*. Computing Systems, vol. 1, no. 4, pages 305–370, Fall 1988. Révisé dans [Rozier 90].

[Rozier 90] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard et W. Neuhauser. *Overview of the CHORUS Distributed Operating Systems*. Rapport technique CS-TR-90-25, Chorus Systems, 1990.

[Rubini 98] Alessandro Rubini. Linux device drivers. O'Reilley & Associates, Inc., Février 1998.

[Rumeur 94] Jean De Rumeur. Communications dans les réseaux de processeurs. Masson, Décembre 1994.

[Sandberg 85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh et Bob Lyon. *Design and Implementation of the Sun Network Filesystem*. In USENIX Association, editeur, Summer conference proceedings, June 11–14, 1985, Portland, Oregon USA, pages 119–130, P.O. Box 7, El Cerrito 94530, CA, USA, Juin 1985.

[Satyanarayanan 85] M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector et M. West. *The ITC Distributed File System : Principles and Design*. Proc. of the 10th ACM Conf. on Operating Systems Principles, vol. 19, no. 5, pages 35–50, Décembre 1985.

[Seznec 96] André Seznec et Fabien Lloansi. *Étude des architectures des microprocesseurs MIPS R10000, UltraSPARC et PentiumPro*. Rapport de recherche RR-2893, INRIA, IRISA - Rennes, Mai 1996.

[Seznec 97] André Seznec et Thierry Lafage. *Évolution des gammes de processeurs MIPS, DEC Alpha, PowerPC, SPARC, x86 et PA-RISC*. Rapport de recherche RR-3188, INRIA, IRISA - Rennes, Juin 1997.

[SNMPv2 W.G. 96a] SNMPv2 W.G., J. Case, K. McCloghrie, M. Rose et S. Waldbusser. *RFC 1905: Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)*, Janvier 1996. Obsoletes RFC1448. Status: DRAFT STANDARD.

[SNMPv2 W.G. 96b] SNMPv2 W.G., J. Case, K. McCloghrie, M. Rose et S. Waldbusser. *RFC 1907: Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)*, Janvier 1996. Obsoletes RFC1450. Status: DRAFT STANDARD.

[Stallings 93] W. Stallings. SNMP, SNMPv2 and CMIP: The practical guide to network-management standards. Addison-Wesley, Reading, 1993.

[Stevens 90] W. Richard Stevens. UNIX network programming. Prentice Hall, 1990.

[Strauss 98] Daryl Strauss et Wook. *Linux Helps Bring Titanic to Life*. Linux Journal, no. 46, Février 1998. Aussi accessible via `http://www.ssc.com/lj/issue46/index.html`.

[Strayer 94] W.T. Strayer, S. Gray et R.E. Cline Jr. *An Object-Oriented Implementation of the Xpress Tranfer Protocol*. In Proceedings of the Second International Workshop on Advanced Communications and Applications for High-Speed Networks (IWACA), Heidelberg, Germany, Septembre 1994. `http://-www.ca.sandia.gov/xtp/SandiaXTP/iwaca2.ps`.

[Sun Microsystems, Inc. 89] Sun Microsystems, Inc. *RFC 1094: NFS: Network File System Protocol specification*, Mars 1989. See also RFC1813. Status: INFORMATIONAL.

[Sun Microsystems, Inc. 90] Sun Microsystems, Inc. *Network Programming Guide*, 1990.

[Talbi 95] El-ghazali Talbi. *Allocation dynamique de processus dans les systèmes distribués et parallèles : état de l'art*. Rapport de recherche LIFL:AS-162, Laboratoire d'Informatique Fondamentale de Lille, Janvier 1995.

[Tanenbaum 95] Andrew S. Tanenbaum. Distributed operating systems. Prentice Hall International Editions, 1995.

[Tanzy 97] Jane-Elise Tanzy. Répartition dynamique de charge dans les réseaux de stations de travail. Mémoire de DEA, Réseaux et Systèmes Distribués (RSD), Université de Nice - Sophia Antipolis, 1997.

[The OMG Consortium 93] The OMG Consortium. *The common object request broker : architecture and specification*. Technical report 90.9.1, Object Management Group, Farmington, MA, USA, 1993. `http://www.omg.org/`.

[Vahalia 96] Uresh Vahalia. UNIX internals: The new frontiers. Prentice-Hall International Editions, 1996.

[van Renesse 96] Robbert van Renesse, Kenneth P. Birman et Silvano Maffeis. *Horus: A Flexible Group Communications System*. Coomunications of the ACM, vol. 39, no. 4, Avril 1996.

[Waldbusser 95] S. Waldbusser. *RFC 1757: Remote Network Monitoring Management Information Base*, Février 1995. Obsoletes RFC1271. Status: DRAFT STANDARD.

[Waldo 94] Jim Waldo, Geoff Wyant, Ann Wollrath et Sam Kendall. *A note on distributed computing*. Rapport technique SMLI TR-94-29, Sun Microsystems Laboratories and Mountain View, CA, Novembre 1994.

[Wang 85] Y. T. Wang et J. T. Morris. *Load Sharing in Distributed Systems*. IEEE Transactions on Computers, vol. C-34, no. 3, pages 204–217, Mars 1985.

[Welsh 97] Matt Welsh, Anindya Basu et Thorsten von Eicken. *ATM and Fast Ethernet Network Interfaces for User-level Communication*. In Proceedings of the 3d Int. Symposium on High Performance Computer Architecture (HPCA), San Antonio, Texas, Février 1997. IEEE.

[XTP Forum 95] XTP Forum. *Xpress Transport Protocol Specification: XTP Revision 4.0*. Rapport technique XTP

95-20, The XTP Forum, 1394 Greenworth Place, Santa Barbara, CA, Mars 1995. Available from http://www.ca.sandia.gov/xtp/biblio.html.

[Yavatkar 96] R. Yavatkar, J. Griffioen et M. Sudan. *A reliable dissemination protocol for interactive collaborative applications*. In The Third ACM International Multimedia Conference and Exhibition (MULTIMEDIA '95), pages 333–344, New York, Novembre 1996. ACM Press.

[Zerrouk 96] Belkacem Zerrouk, Vincent Reibaldi, Frédéric Potter, Alain Greiner, Anne Derieux, Roland Marbot et Reza Nezammzadeh. *RCube : A Message Routing Device Using The OMI/HIC High Speed Link Technology*. In Proceedings of the Third IEEE International Conference on Electronics Circuits and Systems (ICECS'96), Rodos, Grèce, 1996.

[Zhou 92] Songnian Zhou. *LSF: load sharing in large-scale heterogeneous distributed systems*. In Proceedings of the Workshop on Cluster Computing, Tallahassee, FL, Décembre 1992. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.92.

[Zhou 93] Songnian Zhou, Xiaohu Zheng, Jingwen Wang et Pierre Delisle. *Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*. Software – Practice and Experience, vol. 23, no. 12, pages 1305–1336, Décembre 1993.

Résumé

Dans cette thèse, nous nous intéressons aux techniques et outils qui permettent de concevoir et d'optimiser les applications parallèles et réparties sur les réseaux et grappes de stations de travail.

Le premier problème abordé est celui de la répartition dynamique de charge en environnement fortement hétérogène : pour répartir dynamiquement et efficacement la charge d'une application répartie, il faut (i) être en mesure d'évaluer et de comparer la disponibilité des différentes machines du réseau et (ii) savoir mettre ces informations en correspondance avec les besoins en ressources des tâches de l'application.

Pour cela, nous proposons une méthodologie de modélisation empirique du comportement des éléments d'un réseau de stations de travail face à la charge. Cette méthodologie nous permet de construire des indicateurs de charge multi-dimensions et multi-critères. Pour mettre cette méthodologie en pratique, nous avons conçu *LoadBuilder*, une plate-forme répartie d'expérimentation.

Le deuxième problème abordé est celui de l'accès à des mécanismes et protocoles de communication multipoints fiables et ordonnés, à partir d'un système d'exploitation UNIX.

Pour répondre à ce besoin des applications réparties, nous proposons une solution originale, le système de fichiers virtuel MPCFS. Ce système de fichiers permet la création de groupes de communication dynamiques et la réalisation de communications multipoints dans ces groupes, au travers de simples manipulations de fichiers et répertoires. Nous avons développé un prototype de ce système, qui peut être chargé dynamiquement dans le noyau du système Linux 2.0.

Mots-clés: Communications, Répartition dynamique de charge, Hétérogénéité, Protocoles, Réseaux de stations de travail, Systèmes d'exploitation, UNIX, Parallélisme, Système de fichiers.

Abstract

Technics and tools for communications and load balancing in networks of workstations

In this thesis, we focus on technics and tools that help in building and optimizing parallel and distributed applications for networks and clusters of workstations.

The first problem we study is dynamic load balancing in a strongly heterogeneous environment: an efficient dynamic load balancing strategy requires first to evaluate and compare the availability of each of the workstations and second to find a good matching between this availability and the resources requirements of the applications.

For this purpose, we propose an empirical method whose goal is to model the behavior of the components of a network of workstations according to the workload levels being observed. This method allows the construction of multi-dimensional and multi-criteria workload indices. To put this method in practice, we have developed *LoadBuilder*, a distributed environment especially intended to drive distributed experiments.

The second problem we study is accessing reliable ordered multi-point communication mechanisms and protocols in UNIX operating systems.

In order to deal with the communications requirements of distributed applications, we propose an original solution, the MPCFS virtual file system. This file system provides its users a way of (i) creating and managing process groups and (ii) exchanging multi-points messages in these groups. These operations can be achieved easily, through the usual UNIX file and directory operations. We have developed a dynamically loadable module for the Linux 2.0 operating system, that implements a prototype of this file system.

Keywords: Communications, Dynamic Load Balancing, Heterogeneity, Protocols, Networks of workstations, Operating systems, UNIX, Parallel Computing, File system.