N° d'ordre: 3054

THÈSE

Présentée devant

devant l'université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Pascal GALLARD

Équipe d'accueil : PARIS
École doctorale : MATISSE
Composante universitaire : IFSIC/IRISA

Titre de la thèse :

Conception d'un service de communication pour systèmes d'exploitation distribués pour grappes de calculateurs : mise en œuvre dans le système à image unique Kerrighed

soutenue le 21 Décembre 2004 devant la commission d'examen

M. :	Thierry	PRIOL	Président
MM. :	Raymond	NAMYST	Rapporteurs
	Bernard	TOURANCHEAU	
MM. :	Liviu	IFTODE	Examinateurs
	Bertil	FOLLIOT	
	Christine	MORIN	

Marche pas gimli... Anonyme, Mes meilleurs troll

Remerciements

Après trois années de thèse, c'est presque avec soulagement que j'aborde les remerciements liés à mon travail.

Tout d'abord je souhaite vivement remercier les différents membres de mon jury. En particulier je remercie Thierry Priol, Directeur de recherche INRIA à l'IRISA et responsable du projet PARIS qui m'a accueilli, de m'avoir fait l'honneur d'être le président de mon jury. Je remercie Raymond Namyst, Professeur à l'Université de Bordeaux ainsi que Bernard Tourancheau, Sun Microsystems Lab, d'avoir accepté, en dépit d' emplois du temps surchargés, d'être rapporteurs et d'avoir évalué et critiqué le travail réalisé.

Je remercie Christine Morin, directrice de recherche INRIA en charge de l'activité de recherche Kerrighed, de m'avoir encadré et formé au métier de la recherche. Je tiens en particulier à la remercier pour la liberté d'action qu'elle m'a offert. J'ai pu investiguer mes idées et ainsi, en déterminer les limites par moi-même.

Je remercie tout chaleureusement Liviu Iftode, Associate Professor à Rutgers University (NJ, USA), d'avoir accepté d'être examinateur de ma thèse et plus encore de m'avoir offert l'opportunité de travailler dans son équipe. Cette collaboration aura été pour moi très riche d'enseignements scientifique, professionnel et humain.

Je souhaite maintenant remercier les "voisins" tant pour leur soutien moral que gastronomique. Avec André, j'ai débuté la (longue) route vers un lieu étrange, presque mythique, appelé "la recherche". De ces premiers pas hésitants sera née, je l'espère, une belle amitié. Avec Héma et Louis, c'est la magie des pommes du RU se transformant en succulant krumble (avec du sirop d'érable !) que je retiendrai.

Je remercie Gaël, mon camarade historique de bureau, Renaud, mon partenaire d'équitation et Geoffroy, le plus rapide mangeur de crème anglaise de ce côté-ci de la Vilaine, pour les ~~trolls~~ nombreuses discussions qui ont jalonné ma thèse. Les accords des uns et les désaccords des autres m'auront permis de façonner mes idées.

Je remercie David Margerie et Christian Perez pour leurs lumières respectives à des moments où je naviguais dans l'obscurité de la rédaction. Leurs précieuses remarques m'ont permis d'avancer dans l'extraction des idées pertinentes de mon travail.

Je remercie les différents membres du projet IRISA / INRIA PARIS pour la bonne humeur et les très bonnes conditions de travail dont j'ai pu bénéficier. J'attribue une mention spéciale à Élodie Communier pour m'avoir si souvent fait voyager.

Je remercie Florin Sultan, Aniruddha Bohra ainsi que Murali Rangarajan pour l'accueil et la générosité dont ils font preuve depuis le premier jour de mon séjour à Rutgers university.

Enfin, je remercie les personnes sans lequelles, je ne serais certainement pas allé aussi loin dans mon parcours universitaire. En premier lieu, je pense à mes grand-parents, sans qui l'aventure métropolitaine n'aurait certainement pas été possible. La curiosité et la persévérence de Pépé m'auront quotidiennement guidé ces dernières années. Je remercie mes parents pour avoir continué à croire en moi, en des années où la réussite n'était plus au rendez-vous. Enfin, je remercie Magali qui m'accompagne depuis quasiment le début de mon aventure universitaire et qui doit endurer au quotidien ma méthode de travail.

Table des matières

Introduction **11**

1 Introduction aux grappes de calculateurs **17**
1.1 Définitions et propriétés générales 17
1.1.1 Le système d'exploitation 17
1.1.2 Les systèmes de communication 19
1.2 Grappes de calculateurs 21
1.3 Système à image unique 22
1.4 Synthèse 24

2 Environnements d'exécution communicants **25**
2.1 Environnement d'exécution d'applications communiquant par mémoire partagée 26
2.2 Environnement d'exécution d'applications communiquant par message . 26
2.2.1 Primitives de communication de bas niveau 26
2.2.2 Environnement de communication 27
2.3 Recherche de la performance dans les environnements de communication par message 30
2.4 Support au déplacement des processus dans les environnements de communication 30
2.4.1 Tolérance aux fautes dans les environnement de communication par des techniques de point de reprise 32
2.5 Conclusion 34

3 Infrastructure de communication dans les grappes de calculateurs **37**
3.1 Infrastructure matérielle et gestionnaires de communication 37
3.1.1 Système de communication par le SE 38
3.1.2 Accès direct aux matériels de communication 41
3.2 Modèles de communication 43
3.3 Conclusion 49

4 Introduction aux systèmes à image unique **51**
4.1 Classification des systèmes à image unique 51
4.2 Présentation d'un système à image unique : Kerrighed 54

4.3 Gestion globale de la mémoire 54
4.3.1 Architecture de la mémoire 55
4.3.2 Profil de communication 57
4.4 Gestion globale des processus 57
4.4.1 Architecture de la gestion des processus 58
4.4.2 Profil de communication 59
4.5 Gestion globale des synchronisations 60
4.6 Synthèse 60

5 Système de communication pour un système à image unique 63
5.1 Synthèse des communications dans les grappes 64
5.2 Modèle de communication d'un service système distribué 65
5.3 Modèle de communication pour une application communiquant par échange de messages 66
5.4 Transaction de communication 67
5.5 Flux dynamiques 68
5.6 Architecture globale du système de communication 69

6 Architecture de communication pour un système à image unique 71
6.1 Caractéristiques des matériels et des protocoles de communication envisagés 71
6.2 Modèle de communication au sein d'un SSI 73
6.3 Modèle de matériel de communication 75
6.4 Modèle de protocoles de communication 76
6.5 Gestion des propriétés génériques dans l'infrastructure de protocoles . . 78
6.5.1 Gestion de l'identification des nœuds 78
6.5.2 Gestion de la fragmentation 79
6.5.3 Gestion des retransmissions 79
6.5.4 Gestion de l'ordre des messages 79
6.5.5 Modèle d'émission de message 80
6.5.6 Modèle de réception de message 81
6.6 Résumé 83

7 Système de communication pour services distribués 85
7.1 Cadre de l'étude 86
7.2 Concepts généraux liés aux transactions de communication 86
7.3 Les transactions de communication 88
7.3.1 Transaction de communication 88
7.3.2 Élément de transaction 90
7.3.3 Contexte de transaction 91
7.4 Résumé 92

8 Flux de données dynamiques **95**
8.1 Notion de flux de données 95
8.2 Communication intra-grappe de calculateurs 97
8.2.1 Architecture pour la haute performance 97
8.2.2 Introduction aux flux dynamiques 97
8.2.3 Communication directe entre les extrémités distribuées 101
8.2.4 Localisation des extrémités d'un flux 104
8.2.5 Interaction avec les modèles de communication 106
8.2.6 Tolérance aux fautes 109
8.2.7 Intégration dans l'architecture de communication pour SSI . . . 110
8.3 Communication extra-grappe de calculateurs 110
8.3.1 Infrastructure pour l'accès à distance aux structures de données d'un SE 111
8.3.2 Application aux déplacements de sockets TCP/IP 112
8.4 Communication inter-grappes de calculateurs 112
8.5 Résumé 113

9 Éléments de mise en œuvre **115**
9.1 Système de communication de niveau protocole 115
9.2 Interfaces de communication pour les services système distribués 116
9.2.1 Exemple de mise en œuvre d'un service système distribué à l'aide des transactions de communication 116
9.2.2 Interfaces complémentaires 118
9.3 Système de communication par flux dynamiques 119
9.3.1 Communication par pipe 120
9.3.2 Communication par socket unix 120
9.4 Résumé 121

10 Évaluation du système de communication de Kerrighed **125**
10.1 Évaluation du système de communication pour les services système distribués 125
10.2 Évaluation des communications par flux dynamique 126
10.3 Évaluation comparative des performances de communication entre plusieurs SSIs 131
10.4 Synthèse 132

Conclusion **137**

Table des figures

1.1 Modèle de référence OSI 20

2.1 Transmission de messages dans PVM 28
2.2 Communication à l'aide d'intermédiaire pour la tolérance aux fautes . . 34

3.1 Pile réseau d'un noyau Linux, à l'émission 39
3.2 Pile réseau d'un noyau Linux, à la réception 40
3.3 U-Net : limitation du recours au système d'exploitation 42
3.4 Chronogramme d'une communication synchrone 44
3.5 Chronogramme d'une communication asynchrone 45
3.6 Chronogramme d'une communication avec "Madeleine" 46
3.7 Chronogramme de la réception et du traitement d'un RPC 48
3.8 Chronogramme de la réception et du traitement d'un *Active Message* . . 49

4.1 Principe du *home node* 53
4.2 Services distribués de Kerrighed 55
4.3 Automate de transition d'états des pages d'un conteneurs 56
4.4 Empreinte de communication pour une application en mémoire partagée 57
4.5 Gestion globale des processus 58
4.6 Conteneur liant des segments mémoire 59

5.1 Un système de communication pour les applications d'un système à image unique 70

6.1 Système de communication de Kerrighed 74
6.2 Moteurs de communication de Kerrighed 77
6.3 Trame SSI 78
6.4 Émission d'un message 80
6.5 Réception d'un message 81

8.1 Exemple de l'architecture des flux dynamique pour la mise en œuvre de sockets 101
8.2 Flux dynamique 103
8.3 Communication avec un flux dynamique 104
8.4 Protocole de mise à jour des cartes d'extrémités 106

9.1 Interface de programmation pour l'infrastructure de protocole 116
9.2 Réalisation classique de ping-pong 117
9.3 Exemple de ping-pong à l'aide des transactions de communication . . . 117
9.4 Usage classique d'un pipe 120
9.5 Basic pipe implementation 123
9.6 Protocole associé aux sockets Unix 123

10.1 Evaluation du service de ping-pong réalisé à l'aide des transactions de communication 126
10.2 KerNet : Performance locale à un nœud 127
10.3 Performance sur FastEthernet 128
10.4 Performance sur Gigabit Ethernet 129
10.5 Performance sur Myrinet (netdevice interface) 129
10.6 Performance MPI sur FastEthernet 130
10.7 Performance sur Gigabit Ethernet lors d'un déplacement de processus . 131
10.8 Bande passante des *pipe* sans déplacement 133
10.9 Bande passante des *pipe* après un déplacement 133
10.10 Bande passante des *pipe* après deux déplacements 134
10.11 Bande passante des *pipe* après deux déplacements vers le même nœud . 134

Listings

3.1 Interface de communication send/receive 46
3.2 Interface de communication send/receive 47
7.1 Interface de transaction pour les émissions 88
7.2 Interface de transaction pour les réceptions 88
7.3 Descripteur de communication 89
7.4 Interface de gestion des groupes de canaux 89
7.5 Interface de gestion du jeton d'action 90
7.6 Abandon d'une transaction 90
7.7 Interface de transaction pour les emissions 90
8.1 Gestion du jeton de communication 102
8.2 Primitives de manipulation de flux 106
8.3 Primitives d'échange de message 108
9.1 Primitives asynchrones 118
9.2 Interface de communication send/receive 119

Introduction

La conception d'objets manufacturés complexes tels que les satellites, les automobiles, les avions, fait appel à des techniques de simulation numérique. Les modèles étant de plus en plus complets et de plus en plus précis, les besoins en puissance de calcul augmentent sans cesse. Si les machines multiprocesseurs ont longtemps été utilisées pour l'exécution de ces applications à haute performance, les grappes de calculateurs tendent à les détroner : il suffit, pour s'en convraincre, de consulter le *Top500* (*http ://www.top500.org*) qui montre que les grappes représentent désormais 58% des machines les plus puissantes.

Les utilisateurs des machines multiprocesseurs apprécient leur simplicité d'utilisation, d'administration et de programmation dues au fait du regroupement de toutes les ressources de calcul, de mémoire et de stockage au sein d'une unique machine. Il n'en va pas de même pour les grappes de calculateurs. Une grappe de calculateurs est constituée d'un ensemble de machines standard indépendantes, interconnectées par un réseau rapide, dédié à un usage de type serveur de calcul. L'idée d'utiliser les grappes de calculateurs pour l'exécution d'application à haute performance remonte à une dizaine d'années avec le projet Beowulf de la NASA[82]. On ne parlait pas encore de grappe à cette époque mais de *Network Of Workstation* (NOW)[93]. L'intérêt d'une grappe est fondé sur l'espoir de disposer d'une puissance de calcul comparable à la somme des puissances de calcul disponibles dans chacune des machines la composant.

Bien qu'attrayantes de part leur flexibilité et leur coût, les grappes de calculateurs posent un défi quant à leur exploitation. En effet, l'utilisateur et le concepteur d'applications parallèles, historiquement formés aux machines parallèles traditionnelles, ont l'habitude d'interagir avec un système d'exploitation. Ce système d'exploitation offre une abstraction de l'architecture matérielle en vue d'en simplifier l'utilisation. Sur une machine donnée, le système d'exploitation libère le concepteur d'application des contraintes liées au matériel et permet à celui-ci de se concentrer essentiellement sur le cœur de son application. Dans les grappes de calculateurs, chaque nœud exécute sa propre instance d'un système d'exploitation traditionnel. Ces instances étant indépendantes les unes des autres, concepteurs et utilisateurs d'applications pour grappes de calculateurs doivent prendre en compte la présence d'un réseau d'interconnexion dans l'architecture. De ce point de vue, le concepteur d'applications parallèles fait face à une forme de régression par rapport à l'environnement proposé par les machines parallèles.

Afin d'améliorer cet état de fait, des environnements d'exécution communicants ont été proposés. Ils fournissent une vision un peu plus homogène de la grappe de calcula-

teurs. Toutefois, ces environnements se plaçant au-dessus des systèmes d'exploitation, ils ne peuvent complètement masquer l'aspect distribué d'une grappe de calculateurs.

Une voie, explorée depuis plusieurs années maintenant, vise à concevoir un système d'exploitation distribué pour grappe de calculateurs. Ainsi, les systèmes d'exploitation présents sur les machines de la grappe coopèrent pour former le système d'exploitation de la grappe de calculateurs, masquant la distribution des ressources physiques (processeurs, mémoire physique, réseau) sur les différents nœuds. Dans ce but, une abstraction globale aux nœuds de la grappe, pour les différentes ressources disponibles dans la grappe, est réalisée. Un tel système, cherchant à estomper les limites physiques d'un nœud au sein d'un machine virtuelle, est appelé système à image unique.

Objectif de l'étude

Un système d'exploitation pour grappe de calculateurs est constitué d'un ensemble de services distribués mettant en œuvre la gestion globale des ressources de la grappe. Comme dans tout système distribué, les performances de ces services distribués (et donc celles des applications qui les utilisent) dépendent en grande partie des performances du système de communication. Plus globalement, c'est donc du système de communication dont vont dépendre une grande partie des performances et la réactivité globale de la grappe de calculateurs. De plus, dans de nombreuses applications de calcul haute performance, les importants volumes de données entraînent des échanges de messages de grande taille sur le réseau de communication. Les caractéristiques de débit et de latence du système de communication constituent un élément essentiel du système de communication d'un système à image unique destiné à l'exécution d'applications à haute performance.

Avec un système à image unique, les ressources sont gérées globalement à la grappe. En particulier, un système à image unique met en œuvre une stratégie d'équilibrage de charge dynamique pour la gestion globale de la ressource processeur. Ceci implique de pouvoir déplacer un processus d'un nœud vers un autre en cours d'exécution. Pour une application parallèle communiquant par échange de messages, le déplacement d'un des processus de l'application ne doit pas entraîner de dégradation des performances inter-processus. En cas contraire, le bénéfice de l'équilibrage de charge dynamique serait moindre voir nul.

Une grappe étant composée d'un grand nombre de machines, la probabilité de défaillance d'un des nœuds augmente avec la taille de la grappe. Le temps d'exécution d'une application à haute performance (qui peut se compter en jours, en semaine, voire plus) pouvant être supérieur au temps moyen entre défaillances (MTBF) d'une grappe, il est essentiel qu'un système à image unique offre des mécanismes de tolérance aux fautes aux applications parallèles. Les stratégies de recouvrement arrière qui consistent en la sauvegarde périodique de points de reprise, restaurés en cas de défaillance, sont bien adaptées aux applications parallèles à haute performance.

Nos travaux de thèse ont pour objectif la conception et la réalisation d'un système de communication pour l'exécution d'applications à haute performance sur un système à image unique. Il s'agit donc de concevoir un système de communication qui soit

adapté aux besoins des services distribués qui constituent le système à image unique, ainsi qu'à ceux des applications parallèles à haute performance exécutées au-dessus de ce système.

Le système de communication conçu doit garantir la performance des services distribués composant le système à image unique. Il doit également garantir la performance des applications communiquant par échange de messages, y compris lors du déplacement, en cours d'exécution, d'un ou plusieurs processus. Enfin, il doit faciliter la mise en œuvre de stratégies de sauvegarde d'applications parallèles fondées sur le modèle de communication par message.

Il est souhaitable que les services distribués d'un système à image unique soient indépendants de la technologie du réseau d'interconnexion utilisé au sein de la grappe. Le système de communication d'un système à image unique doit donc être suffisamment modulaire pour ne pas imposer de restriction sur le choix de la technologie réseau d'interconnexion. Il est important pour des raisons de performances que le système de communication permette l'exploitation des caractéristiques avancées d'interfaces de communication récentes telles que Myrinet[18]. Enfin, dans un soucis de ré-utilisation de la base logicielle existante, ce système de communication doit permettre l'exécution d'applications conçues pour les environnements communicants de référence (PVM, MPI) sans modification des codes sources ou des binaires associés.

Le système de communication constitue le cœur d'un système à image unique pour grappe de calculateurs. Notre contribution comporte plusieurs volets. En premier lieu, nous proposons une architecture de système de communication pour système à image unique permettant le multiplexage de l'accès aux ressources physiques pour les communications générées par le système d'exploitation et celles générées par les applications communicantes.

Sur cette base commune sont développés un mécanisme efficace d'appel de procédure à distance à l'usage des services système distribués, appelé **transaction de communication**, et un mécanisme de **flux dynamique** permettant la réalisation d'une version mobile de l'ensemble des interfaces de communication conventionnelles utilisées par les applications. Par ailleurs, le système de communication proposé a été conçu pour permettre la réalisation d'un environnement logiciel pour la haute disponibilité du système d'exploitation pour grappe de calculateurs.

Le système de communication que nous avons défini, appelé KerNet, a été mis en œuvre dans le système à image unique Kerrighed, conçu et réalisé au sein du projet INRIA PARIS de l'IRISA. Une évaluation des performances des mécanismes proposés a été menée sur différents réseaux d'interconnexion (Ethernet, Gigabit Ethernet et Myrinet).

Le système de communication KerNet a été intégré aux versions `1.0-rc` de Kerrighed qui est diffusé sous forme de logiciel libre avec une licence GPL sur le site : *http ://www.kerrighed.org*. Depuis novembre 2004, le système Kerrighed est distribué dans la suite OSCAR (*http ://ssi-oscar.irisa.fr*). Des applications parallèles réelles, notamment MPI, fournies par des partenaires industriels ont été exécutées avec succès au-dessus de versions de Kerrighed intégrant nos travaux, dans le cadre du PEA ARCO-

SIM COCA de la DGA, mené en partenariat avec Cap Gemini et l'ONERA-CERT et dans le cadre d'une collaboration avec EDF R&D. Kerrighed a également été déployé sur des grappes à l'ORNL et à l'université d'Ulm en Allemage à des fins d'expérimentation et d'évaluation.

Plan du document

La suite du document comporte dix chapitres.

Nous présentons tout d'abord un état de l'art sur les travaux dans le domaine des systèmes de communication pour grappe de calculateurs. Après avoir énoncé quelques définitions et propriétés relatives aux systèmes à image unique (chapitre 1), le chapitre 2 présente les fonctionnalités des environnements d'exécution conçus pour des applications parallèles communiquant par message. Le chapitre 3 décrit quelques unes des technologies de communication les plus fréquemment employées comme système d'interconnexion dans les grappes de calculateurs ainsi que différents modèles de programmation des systèmes de communication. Nous abordons dans le chapitre 4 les systèmes à image unique, en dégageant les besoins en terme de communication.

Dans la seconde partie, introduite par le chapitre 5, nous présentons nos contributions. Le chapitre 6 présente l'architecture globale du système de communication que nous proposons pour un système à image unique. Nous présentons ensuite nos deux contributions essentielles : le concept de transaction de communication pour la conception de services système distribués réactifs (chapitre 7) et le concept de flux dynamique qui offre un support adapté à l'exécution des applications communicantes par échange de messages dans le cadre d'un système à image unique (chapitre 8). Enfin, nous donnons quelques éléments de mise en œuvre dans le chapitre 9, avant de présenter dans le chapitre 10 les résultats de l'évaluation de performances que nous avons menée.

Le dernier chapitre de ce document est consacré à un bilan de nos travaux et à une présentation des perspectives ouvertes par ces derniers.

Publications

Les travaux présentés dans ce manuscrit ont fait l'objet de diverses publications, à la fois dans des journaux, et dans des conférences internationales et nationales. Ces publications concernent :

- les communications internes à une grappe de calculateurs [1, 3, 10],
- le système Kerrighed [2, 5, 6, 7, 11],
- la notion de service distribué [4, 9, 13, 16],
- les communication extérieures à une grappe ainsi que la haute disponibilité des services mis en œuvre sur grappe [8, 12, 14, 15].

Articles dans des revues internationales

1 **Dynamic Streams For Efficient Communications between Migrating Processes in a Cluster.** Pascal Gallard et Christine Morin. *Parallel Processing*

Letters, 13(4), Décembre 2003. http ://www.inria.fr/rrrt/rr-4987.html

2 **Towards an Efficient Single System Image Cluster Operating System.** Christine Morin, Pascal Gallard, Renaud Lottiaux, et Geoffroy Vallée. *Future Generation Computer Systems*, 20(2), Janvier 2004.
http ://www.irisa.fr/paris/Biblio/Papers/Morin/MorGalLotVal03FGCS.pdf

Articles dans des conférences internationales

3 **Dynamic Streams For Efficient Communications between Migrating Processes in a Cluster**. Pascal Gallard et Christine Morin. In *Euro-Par 2003 : Parallel Processing*, volume 2790 of Lect. Notes in Comp. Science, Klagenfurt, Austriche, pages 930–937, Août 2003. Springer-Verlag.
http ://www.irisa.fr/paris/Biblio/Papers/Gallard/GalMor03europar.ps

4 **Dynamic Resource Management in a Cluster for High-Availability.** Pascal Gallard, Christine Morin, et Renaud Lottiaux. In *Euro-Par 2002 : Parallel Processing*, volume LNCS 2400, Paderborn, Allemagne, pages 588–592, Août 2002. Springer-Verlag.
http ://www.irisa.fr/paris/Biblio/Papers/Gallard/GalMorLot02europar.ps

5 **Kerrighed : a Single System Image Cluster Operating System for High Performance Computing.** Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, et Louis Rilling. In *Proc. of Europar 2003 : Parallel Processing*, volume 2790 of Lect. Notes in Comp. Science, pages 1291–1294, Août 2003. Springer Verlag.
http ://www.irisa.fr/paris/Biblio/Papers/Morin/MorLotValGalUtaBAdRil03↩
europar.pdf

6 **Towards an Efficient Single System Image Cluster Operating System.** Christine Morin, Pascal Gallard, Renaud Lottiaux, et Geoffroy Vallée. In *Proc. of Intl. Conf. on Architecture and Algorithms for Parallel Processing (ICA3PP 2002)*, Beijing, Chine, pages 370–377, Octobre 2002. IEEE Computer Society.
http ://www.irisa.fr/paris/Biblio/Papers/Morin/MorGalLotVal02ica3pp.ps.gz

7 **Kerrighed and Data Parallelism : Cluster Computing on Single System Image Operating Systems.** Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, David Margery, Jean-Yves Berthou et Isaac Scherson. In *Proc. of Intl. Conf. on Cluster Computing (CLUSTER 2004)*. À paraître.

8 **Remote Repair of Operating System State Using Backdoors**. A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, et L. Iftode. *International Conference on Autonomic Computing (ICAC-04)*, New-York, NY, Mai 2004.

Articles dans des conférences nationales

9 **Gestion dynamique de services dans une grappe de calculateurs.** Pascal Gallard, Christine Morin, et Renaud Lottiaux. In *Journées des jeunes chercheurs en systèmes d'exploitation (ASF)*, Hammamet, Tunisie, pages 485–592, Avril 2002.

http ://www.irisa.fr/paris/Biblio/Papers/Gallard/GalMorLot02renpar.ps

Rapports de recherche

10 **Dynamic Streams for Efficient Communications Between Migrating Processes in a Cluster.** Pascal Gallard et Christine Morin. Research Report RR-4987, INRIA, IRISA, Rennes, France, Novembre 2003.
http ://www.inria.fr/rrrt/rr-4987.html

11 **Towards an Efficient Single System Image Cluster Operating System.** Christine Morin, Pascal Gallard, Renaud Lottiaux, et Geoffroy Vallée. Research report RR-5018, INRIA, IRISA, Rennes, France, Décembre 2003.
http ://www.irisa.fr/bibli/pi/2003/1578/1578.html

12 **System Support for Nonintrusive Failure Detection and Recovery using Backdoors**. F. Sultan, A. Bohra, P. Gallard, I. Neamtiu, S. Smaldone, Y. Pan, et L. Iftode. Technical report DCS-TR-524, Rutgers University, Décembre 2003.
http ://discolab.rutgers.edu/bda/remrecov04.pdf

13 **Dynamic Resource Management in a Cluster for Scalability and High-Availability.** Pascal Gallard, Christine Morin, et Renaud Lottiaux. Research Report 4347, INRIA, Janvier 2002.
http ://www.inria.fr/rrrt/rr-4347.html

14 **Citadel : Defensive Architectures for Computer Systems and Networks**. F. Sultan, A. Bohra, P. Gallard, S. Smaldone, Y. Pan, B. Nath et L. Iftode. Technical report DCS-TR-554, Rutgers University, Mars 2004.
http ://discolab.rutgers.edu/bda/DCS-TR-554.ps

Divers

15 **Backdoors : A System Architecture for Nonintrusive Remote Healing.** Florin Sultan, Aniruddha Bohra, Pascal Gallard, Iulian Neamtiu, Steve Smaldone, Yufei Pan, et Liviu Iftode. Poster presented at *the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Octobre 2003.
http ://discolab.rutgers.edu/bda/bd_poster.gif

16 **Haute disponibilité dans les grappes de calculateurs.** Pascal Gallard. Rapport de DEA (2001).

Chapitre 1

Introduction aux grappes de calculateurs

1.1 Définitions et propriétés générales

1.1.1 Le système d'exploitation

Définition 1 ***Système d'exploitation – SE****: Logiciel réalisant une abstraction des ressources physiques de la machine sur laquelle il s'exécute.*

L'abstraction d'une **ressource physique** permet d'offrir un ensemble de **ressources logiques** associé à la ressource physique. Ainsi, le partage de la ressource physique est réalisé par le SE entre les ressources logiques offertes. Par exemple, la mémoire physique d'une machine est composé de pages mémoire appelées *cadre de page*. Les systèmes d'exploitation modernes fournissent tous un mécanisme de mémoire virtuelle. Une page de mémoire virtuelle est temporairement associée à un cadre de page physique. À tout moment, le SE d'exploitation peut décider de déplacer la page virtuelle vers un autre cadre de page, ou vers un autre support de donnée afin de libérer un espace physique pour une autre page virtuelle. Le SE peut donc fournir plus de pages virtuelles qu'il n'en dispose physiquement.

Propriété 1 ***Transparence de l'accès aux ressources physiques****: Nous appelons* ***transparence de l'accès aux ressources physiques*** *le fait d'exploiter une ressource quelconque sans pré-requis concernant la localisation physique de cette ressource.*

Un SE peut rendre transparent l'usage d'une ressource en fournissant une interface d'utilisation de cette ressource. Une telle interface permet à une application quelconque d'être indépendante de la technologie (matérielle ou logicielle) employée par la ressource. En particulier, si l'interface des fonctions fournies par le SE reste inchangée, les applications ont une compatibilité binaire avec les évolutions du SE.

Propriété 2 ***Multiplexage de l'accès aux ressources physiques****: Nous appelons* ***multiplexage de l'accès aux ressources physiques*** *le fait de permettre à plusieurs logiciels d'accéder simultanément aux mêmes ressources physiques.*

À ce niveau, nous ne faisons aucune hypothèse particulière sur le logiciel. Le multiplexage peut notamment s'effectuer entre des logiciels faisant partie du SE et d'autres faisant partie d'applications utilisateurs.

Définition 2 ***Service système****: Nous appelons* ***service système*** *une unité logique au sein du SE participant au fonctionnement du SE.*

Les principaux services systèmes dans un SE classique sont la gestion de la mémoire, celle des processus, celle du système de fichiers, et celle des communications.

Définition 3 ***Fil d'exécution****: Nous appelons* ***fil d'exécution*** *toute succession d'instructions exécutées par un processeur ayant une unité logique d'action.*

Cette définition correspond au cas général d'une exécution de code pour un processeur. En particulier, cette définition ne prend pas en compte le contexte de l'exécution ni même l'élément déclencheur de cette exécution. En effet, dans un système d'exploitation de type Unix, de nombreux fils d'exécution sont disponibles à tout instant pour le processeur. Le choix du fil à exécuter est dévolu à un ordonnanceur.

Définition 4 ***Processus****: Nous appelons* ***processus*** *des fils d'exécution pouvant être manipulés par l'ordonnanceur du système d'exploitation.*

Par cette définition, nous mettons délibérement l'accent sur le fait qu'un fil d'exécution puisse être momentanément stoppé de sa propre initiative, à la suite d'une opération système spécifique, ou repris à la suite d'une décision de l'ordonnanceur.

Définition 5 ***Processus en espace utilisateur****: Un* ***processus en espace utilisateur*** *est un processus exécuté sans privilège particulier.*

Un processus utilisateur s'exécute sans droit privilégié. En particulier, un processus utilisateur ne peut accéder qu'à son propre espace d'adressage et ne peut parcourir les structures privées du système d'exploitation. Les interactions des processus utilisateurs avec le système d'exploitation se font au travers d'un mécanisme dédié, appelé les **appels systèmes**. Les appels systèmes réalisent notamment le changement de contexte nécessaire à l'exécution de code interne au système d'exploitation.

Définition 6 ***Processus en espace noyau****: Un* ***processus en espace noyau*** *est un programme exécuté dans le contexte du système d'exploitation.*

Le contexte de systène d'exploitation permet à un processus noyau de disposer de droits privilégiés. En particulier, un tel processus peut accéder et modifier toutes les parties (internes ou publiques) du système d'exploitation. Toutefois, comme son nom le suggère, un processus noyau peut voir son exécution momentanément interrompue et reprise ultérieurement (sur décision de l'ordonnanceur).

L'ordonnanceur partage l'usage de la ressource processeur entre les différents processus selon la politique retenue par l'administrateur de la machine. Toutefois, les processus

(selon notre définition) ne disposent pas d'un usage exclusif de la ressource processeur. En effet, de nombreuses architectures matérielles permettent d'associer une ou plusieurs actions à un événement d'origine matérielle (mécanisme de type *interruption*).

Définition 7 ***Fil d'exécution en interruption****: Nous appelons* ***fil d'exécution en interruption*** *la succession d'opérations réalisées à la suite d'une interruption.*

Ce fil d'exécution dispose des droits privilégiés nécessaires aux accès à l'ensemble de l'espace d'adressage correspondant à la mémoire physique de la machine. En particulier, il est possible de consulter ou modifier les parties du système d'exploitation présentes dans ces zones. Par ailleurs, de part sa nature fortement liée au matériel, un fil d'exécution en interruption dispose d'un contexte d'exécution ne lui permettant pas d'être manipulé par l'ordonnanceur. En particulier, un tel fil ne peut pas être momentanément arrêté puis repris. En contre-partie, le démarrage d'un fil d'exécution en interruption est immédiatement consécutif à l'événement ayant généré cette interruption en interrompant tout fil d'exécution présent sur le processeur. Ce point constitue la différence fondamentale entre les fils d'exécution en interruption et les processus : la reprise d'un processus dépend du choix de l'ordonnanceur.

1.1.2 Les systèmes de communication

Définition 8 ***Système de communication – SC****: L'ensemble des éléments de communication (physiques ou logiciels) participant à l'échange de données est appelé* ***système de communication****.*

Les systèmes de communication comprennent le matériel de communication, les gestionnaires associés aux matériels ainsi que les bibliothèques de communication fournissant les interfaces de programmation. Un modèle a été spécifié afin de décrire les systèmes de communication. Ce modèle est appelé **modèle de référence OSI** (*Open System Interconnection*). Ce modèle décomposé en sept couches est représenté sur la figure 1.1. Dans la suite de ce chapitre, nous ne présentons que certaines des couches du modèle OSI.

Définition 9 ***Technologie de communication****: Nous appelons* ***technologie de communication*** *l'ensemble des caractéristiques et propriétés définissant un matériel de communication.*

Les technologies de communication correspondent au niveau physique du modèle OSI. Ce niveau n'offre pratiquement aucune abstraction aux informations échangées entre les matériels de communication. Les informations échangées sont des bits, découlant directement des signaux électriques échangés.

Les technologies de communication sont couramment classées en fonction des distances couvertes.

Définition 10 ***Wide Area Network – WAN****: Les réseaux interconnectant des machines ou des sites géographiquement séparés par de grandes distances (supérieures à quelques kilomètres) sont appelés* ***WAN***

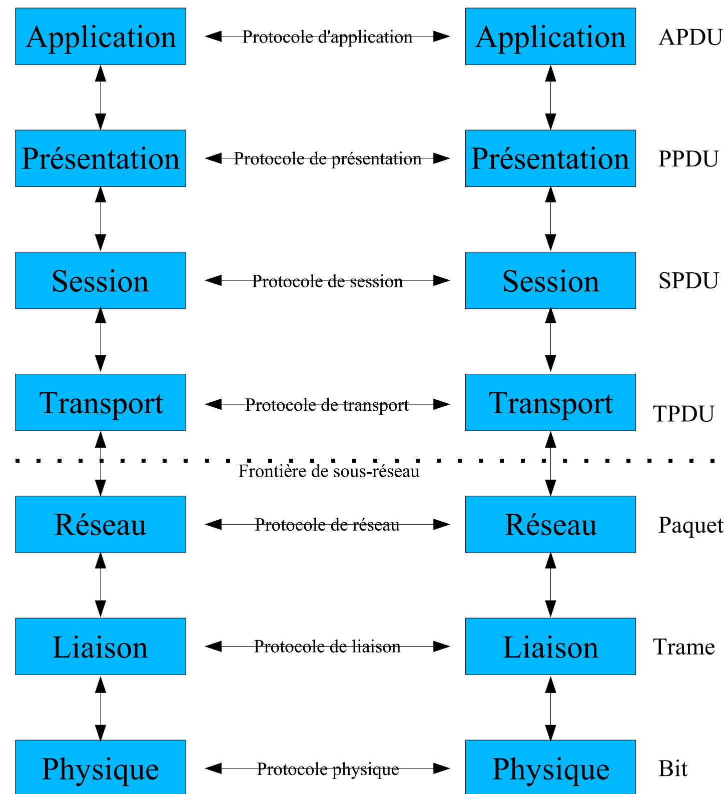


FIG. 1.1 – Modèle de référence OSI

.

ATM[49], FDDI[44] ou *Frame Relay*[28] sont des exemples de technologies de type *WAN*.

Définition 11 ***Local Area Network – LAN***: *Les réseaux interconnectant des machines géographiquement proches, à l'échelle d'un bâtiment, sont appelés LAN.*

L'exemple probablement le plus célèbre actuellement est l'Ethernet[78] qui autorise des distances de l'ordre de quelques centaines de mètres.

Définition 12 ***System Area Network – SAN***: *Les réseaux interconnectant des machines très proches et exploités exclusivement pour les communications entre ces machines sont appelés des SAN.*

Myrinet[18], VIA[23], InfiniBand[8], QsNet[67] et SCI[63] sont des exemples de *SAN*.

En parallèle au classement selon les distances effectué précédemment, nous pouvons effectuer un classement selon l'hétérogénéité des systèmes. En particulier les *SAN* correspondent fréquemment à l'interconnexion de machines semblables sans l'usage de

matériels spécifiques tels que les passerelles et les ponts de communication. L'interconnexion entre des *WAN* ou des *LAN* se fait fréquemment à l'aide du protocole IP[85, 86, 87]. À l'opposé, les *SAN* utilisent presque systématiquement un protocole spécifique et éventuellement proposent une passerelle vers d'autres protocoles.

La simplification de l'architecture réseau permet l'amélioration des performances globales notamment en matière de latence et de bande passante.

Propriété 3 ***Latence:*** *Temps écoulé entre l'émission d'un message et sa réception (effective).*

Propriété 4 ***Bande passante:*** *La bande passante d'une liaison est le débit maximal entre deux extrémités de cette liaison.*

1.2 Grappes de calculateurs

Définition 13 ***Réseau de stations de travail:*** *Un réseau de stations de travail (**Network of Workstations – NOW**[93]) est un ensemble de machines quelconques interconnectées par un réseau local (LAN).*

Une machine employée dans un *NOW* peut avoir un utilisateur attitré ayant priorité sur la machine. Dans ce contexte, le *NOW* est vu comme un outil d'exploitation des ressources inutilisées de la machine en l'absence de son utilisateur principal. En cas de retour de celui-ci, les ressources occupées de sa machine (principalement processeur et mémoire) doivent être restituées.

Définition 14 ***Grappe de calculateurs:*** *Une **grappe de calculateurs** est un ensemble de machines indépendantes, interconnectées par un réseau physique à haute-performance, et utilisées comme une ressource de calcul unique.*

Définition 15 ***Nœud d'une grappe de calculateurs:*** *Une machine quelconque d'une grappe de calculateurs est appelée **nœud** de cette grappe.*

Dans le contexte des grappes de calculateurs, les nœuds sont considérés en "salle blanche", sans utilisateurs *a priori*. Les grappes de calculateurs n'ont pas la contrainte de devoir libérer un nœud à un moment ou un autre (sauf maintenance prévue par les administrateurs de la grappe).

La première ressource au sein d'une grappe de calculateurs réside dans les processeurs. Le problème des gestionnaires d'une telle machine est d'optimiser l'usage de cette ressource disséminée dans plusieurs machines physiquement distinctes.

Propriété 5 ***Placement de processus à distance:*** *Le **placement de processus** correspond à la phase de déploiement des processus d'une application parallèle, sur différents nœuds d'une grappe, en fonction des ressources disponibles dans la grappe de calculateurs.*

Le placement de processus est l'outil principal dans la gestion des grappes de calculateurs. Les systèmes de gestion par lot d'applications permettent d'améliorer l'usage des grappes de calculateurs en fonction des politiques propres au calculateur. De tels systèmes permettent d'ordonner les tâches (i.e : l'application ainsi que l'ensemble de paramètres d'exécution), préalablement soumises, de manière à optimiser l'usage de la grappe selon les critères retenus par ses administrateurs.

Propriété 6 ***Déplacement de processus****: Possibilité d'interrompre l'exécution d'un processus sur un nœud et de la poursuivre sur un autre nœud de la grappe.*

Certaines tâches soumises peuvent nécessiter l'usage de la grappe de calculateurs pendant une longue période. Pendant cette période, les conditions d'utilisation de la grappe peuvent être modifiées :

- des ressources se libèrent, et pourraient être affectées à d'autres tâches en cours d'exécution,
- une tâche de priorité supérieure est soumise et requiert tout ou partie des ressources déjà utilisées,
- au sein d'une tâche, une affinité est découverte entre un processus et une ressource disponible sur un nœud en particulier.

Lors de telles modifications, le déplacement de processus permet de re-déployer l'ensemble des tâches pour répondre aux décisions de l'ordonnanceur global de la grappe de calculateurs.

1.3 Système à image unique

Les premiers réseaux de stations de travail se caractérisent par un ensemble de machines hétéroclites du point de vue du matériel et du logiciel. Dans de telles architectures, la priorité est placée sur l'usage des machines par leur utilisateur habituel. Ces machines n'intègrent réellement la grappe de calculateurs que dans les moments d'absence de l'utilisateur. Selon le profil d'utilisation de la machine, les caractéristiques matérielles et logicielles (y compris du système d'exploitation) peuvent différer d'une machine à l'autre. Cette hétérogénéité fait qu'il était difficilement envisageable de considérer une telle architecture comme une machine parallèle bien définie.

Définition 16 ***Grappe de calculateurs homogène****: Nous appelons grappe de calculateurs homogène, une machine dont les nœuds sont physiquement (notamment pour les systèmes de communication) équivalents et dont la base logicielle est identique (système d'exploitation, bibliothèques et applications).*

L'usage et la programmation d'une grappe de calculateurs ne sont généralement pas des opérations aisées. La principale difficulté est liée à la notion de nœud dans la grappe. Les utilisateurs doivent garder à l'esprit qu'une grappe est composée de nœuds distincts cohabitant pour la réalisation d'une tâche particulière. Afin d'estomper ces notions matérielles, des infrastructures logicielles ont été proposées afin de gérer globalement la grappe de calculateurs.

Définition 17 ***Single System image – SSI****: Nous appelons système à image unique (**SSI**) un ensemble logiciel créant à partir d'une grappe de calculateurs, une machine virtuelle unique.*

Dans la suite de ce document, nous nous intéressons aux SSIs ayant comme modèle de machine virtuelle celui des machines multi-processeurs à mémoire partagée (type *SMP*).

Propriété 7 ***Transparence d'architecture matérielle****: Nous appelons* ***transparence d'architecture matérielle*** *le fait qu'une application ne puisse pas distinguer si elle est exécutée sur une véritable machine SMP ou sur un système à image unique.*

Cette propriété permet à une application conçue pour une machine de type SMP d'être déployée sur une grappe de calculateurs sans modification de son code source, sans même une nouvelle édition de lien. Elle permet la compatibilité binaire entre les architectures SMP et les architectures système à image unique.

Propriété 8 ***Transparence de la localisation physique d'une ressource****: Nous appelons* ***transparence de la localisation physique d'une ressource*** *le fait qu'un processus puisse exploiter localement une ressource logique indépendamment de la localisation physique de cette ressource.*

Il s'agit là des versions adaptées aux systèmes à image unique de la propriété de transparence d'accès aux ressources physiques au sein d'un SE pour machine simple.

Définition 18 ***Service distribué****: Un* ***service distribué*** *est un ensemble logiciel présent sur l'ensemble des nœuds de la grappe de calculateurs afin d'offrir une gestion globale et cohérente d'une ressource au sein de la grappe.*

Les services distribués peuvent être attachés à la gestion de ressources physiques telles que la mémoire ou les disques, mais peuvent également offrir une gestion globale d'entités logiques telles que les processus, les identifiants uniques, etc. Un SSI est généralement constitué de plusieurs services distribués.

De part son architecture, une grappe de calculateurs est composée d'un grand nombre de nœuds. Ce grand nombre de nœuds augmente la probabilité de défaillance au sein de la grappe. De plus, pour des raisons de gestion, un nœud peut être ajouté, retiré ou tout simplement remplacé au sein de la grappe.

Propriété 9 ***Changement dynamique de configuration****: Nous appelons* ***changement dynamique de configuration*** *la prise en compte de l'ajout, du retrait ou du remplacement de tout élément physique de la grappe de calculateurs.*

En particulier, nous nous intéressons aux changements en relation avec les liaisons de communication ainsi qu'à ceux en relation avec les nœuds de la grappe.

Propriété 10 ***Haute disponibilité du système:*** *La* ***haute disponibilité du système*** *correspond au fait que le système d'exploitation reste opérationnel et fiable malgré les changements de configuration ou les défaillances au sein de la grappe.*

Propriété 11 ***Tolérance aux fautes des applications:*** *Permettre la poursuite de l'exécution d'une application malgré la perte d'un nœud possédant une partie de l'application (processus, espace mémoire, etc.).*

1.4 Synthèse

Un système d'exploitation permet le partage et la gestion des ressources disponibles sur une machine. Avec l'apparition des grappes de calculateurs, un nouveau type de système d'exploitation est apparu. Ces systèmes d'exploitation distribués ont pour objectif de partager et de gérer efficacement l'ensemble des ressources disponibles au sein de la grappe de calculateurs. La gestion globale des ressources se fait au travers d'un ensemble de services systèmes distribués spécialisés dans chacune des ressources disponibles de la grappe. L'objectif de certains systèmes à image unique est d'offrir une machine à mémoire partagée (de type SMP) au-dessus d'une grappe de calculateurs.

Pour l'utilisateur d'une telle machine, le service système central est celui de la gestion des ressources processeur. En effet, des outils de placement et de déplacement de processus d'applications parallèles doivent être disponibles. Ces outils sont nécessaire pour permettre l'usage de politique efficace dans la gestion de la ressource processeur, à l'instar de ce qui existe pour les machines parallèles traditionnelles. De telles politiques sont mises en œuvre grâce à des primitives telles que le déplacement des processus. Par ailleurs, la gestion globale des ressources est assurée par des services distribués dont les performances globales influent sur celles du SSI complet.

Du fait de son architecture distribuée, et du nombre de machines pouvant composer une grappe de calculateurs, le problème de la défaillance d'un nœud pendant l'exécution d'une application est réel. La haute disponibilité du système est par conséquent un élément essentiel des système à image unique et des outils de tolérance aux fautes doivent être fournis aux applications.

De par son architecture, une grappe de calculateurs est fortement liée aux caractéristiques de son système de communication et en particulier de son réseau d'interconnexion. Par conséquent, la conception de système à image unique offrant l'ensemble des propriétés précédemment évoquées doit se faire en parallèle avec la conception d'un système de communication adapté aux contraintes des communications internes au système à image unique, et pas seulement interne aux applications parallèles.

Chapitre 2

Environnements d'exécution communicants

Les grappes de calculateurs représentent une architecture distribuée. Par conséquent, les échanges de messages sur le réseau d'interconnexion sont obligatoires pour toute application parallèle déployée sur plusieurs nœuds.

Dans un systèmes à image unique les applications peuvent être programmées selon deux modèles distincts : la programmation par mémoire partagée et la programmation par échange de messages. Nous écartons le cas des applications hybrides. Une application hybride est une application exploitant à la fois un modèle de mémoire partagée pour certains calculs, et un modèle par échange de messages pour d'autres types d'opérations. Par définition, une application hybride est (sous l'angle des communications) la combinaison des caractéristiques de chacun des deux modèles déjà cités.

Dans la suite de ce chapitre, nous abordons tout d'abord très brièvement les environnements d'exécution d'applications parallèles communiquant par mémoire partagée. Dans un second temps, nous étudions de manière détaillée les environnements d'exécution d'application parallèle communiquant par échange de messages.

Notre étude des environnements d'exécution d'application communiquant par échange de messages est ciblée sur les aspects de communication. Plus précisément, nous nous intéressons à la façon dont sont mises en œuvre, dans ces environnements, les propriétés qui nous semblent souhaitables d'offrir aux applications fondées sur le modèle de communication par message dans les systèmes à image unique. Le paragraphe 2.5 résume les points clés du chapitre.

Ce chapitre ne constitue pas une étude exhaustive de ces problématiques. Au contraire, il s'agit de présenter certains des travaux les plus significatifs ayant traité de ces différents aspects.

2.1 Environnement d'exécution d'applications communiquant par mémoire partagée

Le modèle de programmation à mémoire partagée est issu des machines parallèles à mémoire partagée. Dans ce modèle, les différents processus d'une application parallèle partagent une mémoire commune. Lorsqu'un processus a besoin d'une donnée, il lui suffit de consulter son espace mémoire. Les problèmes de synchronisation peuvent être résolus par l'emploi de verrous directement contenus dans la mémoire partagée. Il n'est donc pas nécessaire d'utiliser une autre forme de communication entre les processus.

Dans le monde des machines parallèles, le partage de la mémoire est réalisé de manière matérielle par la machine. Dans le monde des grappes de calculateurs, il est nécessaire de simuler le partage de la mémoire à l'aide d'un système de mémoire partagée répartie (*Distributed Shared Memory* –DSM[51]). Les DSM ont été développées en tant qu'extension du système d'exploitation (SCIOS[45]) ou plus fréquemment en tant qu'intergiciel (Shasta[72], Blizzard-S[74], Mome[37]). À l'aide d'un tel système, des applications exploitant le principe de la mémoire partagée peuvent être déployées et exécutées sur des grappes de calculateurs sans modification majeure de leur modèle de programmation .

Par conséquent, l'ensemble des communications sur le réseau d'interconnexion n'est pas directement réalisé par l'application mais par le système DSM suite aux défauts de page de celle-ci. Ce principe reste vrai, lorsque pour des raisons de performance, le système de verrou est pris en charge par un service dédié et non plus directement par la DSM : les communications par le réseau d'interconnexion, sont masquées par l'environnement de la grappe de calculateurs (systèmes d'exploitation ou intergiciel spécifique).

Dans le reste du chapitre, nous nous intéressons aux modèles de programmation fondés sur des communications explicites.

2.2 Environnement d'exécution d'applications communiquant par message

À la différence du modèle précédent, le concepteur d'une application parallèle peut vouloir contrôler lui-même les échanges de messages sur le réseau d'interconnexion. Pour cela, il peut choisir de programmer à partir de primitives de communication de bas niveau, ou au contraire de s'appuyer sur des environnements de communication existants.

2.2.1 Primitives de communication de bas niveau

Définition 19 ***Primitive de communication****: Nous appelons* ***primitive de communication*** *un outil logiciel ne traitant que la problématique des communications.*

Les primitives de communication se caractérisent par un service entièrement axé sur un modèle de communication. Les primitives de communication correspondent :

– aux interfaces de programmation fournies pour le matériel d'interconnexion choisi : GM[59, 60], MX[61] pour Myrinet, SCICI[34] pour SCI, *IB access*[69] pour Infiniband, *etc.*
– aux protocoles de communication, par exemple l'interface *socket*[83].

Une étude des propriétés liées aux performances de ces primitives de communication est réalisée dans le chapitre suivant.

Chacune de ces primitives de communication est spécifique à un matériel ou un protocole de communication. Par conséquent, une application parallèle utilisant de telles primitives, est spécifiquement écrite pour un matériel ou pour un protocole de communication donné. L'intérêt d'utiliser de telles primitives de communication est :
– de pouvoir mettre en œuvre un modèle de communication très spécifique non offert par un environnement de communication,
– d'espérer bénéficier des meilleurs performances possibles (en terme de bande passante ou de latence) en réduisant les intermédiaires entre l'application et le système de communication.

Considérons maintenant la propriété de calcul de point de reprise et celle du changement dynamique de configutation de la grappe de calculateurs. Le support de ces propriétés dans des applications exploitant des primitives de communication de bas niveau ne peut être réalisé que dans l'application elle-même. Un tel ajout présente deux inconvénients. D'une part, il est réalisé application par application. D'autre part, le code utile (pour fournir le résultat attendu) de l'application est noyé dans le code gérant le système de communication et les propriétés associées.

2.2.2 Environnement de communication

Comme souligné précédemment, le foisonnement des technologies de communication a entraîné la création d'un grand nombre d'interfaces de communication spécifiques ainsi que de protocoles de communication. De plus, ces primitives de communication ne fournissent pas toutes le même niveau d'abstraction des communications, ni même des fonctionnalités équivalentes. Par conséquent, la réalisation d'une application avec de telles primitives limite grandement leur portabilité sur d'autres architectures de communication. C'est en partie pour répondre au problème de l'hétérogénéité des matériels de communication que des environnement de communication ont été créés.

Définition 20 ***Environnement de communication****: Nous appelons* ***environnement de communication*** *un intergiciel offrant, à partir de primitives de communication variées, un modèle commun des communications ainsi qu'un modèle commun d'exécution.*

Architectures de type parallèle Les ancêtres des environnements de communication sont des outils comme la bibliothèque P4[22] ou PARMACS[66]. P4 est un ensemble de macros et de fonctions, développé par le *Argonne National Laboratory* afin d'aider au développement d'applications distribuées indépendantes de l'architecture matérielle

(principalement des machines parallèles). Ainsi, P4 fournit notamment les briques de base pour émettre et recevoir des données et créer des processus sur des nœuds distants.

L'aboutissement des abstractions successives des systèmes de communication ainsi que des modèles d'exécution aura été le système PVM[92] qui propose un modèle d'abstraction de la machine elle-même. À partir d'un fichier de configuration, PVM déploie un ensemble de processus (appelés *daemons*) représentant la machine parallèle. Lorsqu'une application écrite pour PVM lui est soumise, les *daemons* créent les processus de l'application parallèle et en assurent le contrôle. Les communications entre processus de l'application sont interceptées et transmises par les *daemons* (figure 2.1). Dans cet exemple, afin de communiquer avec le processus applicatif **P3**, les processus **P1** et **P2** passent par l'intermédiaire des *daemons* **D1** et **D2**.

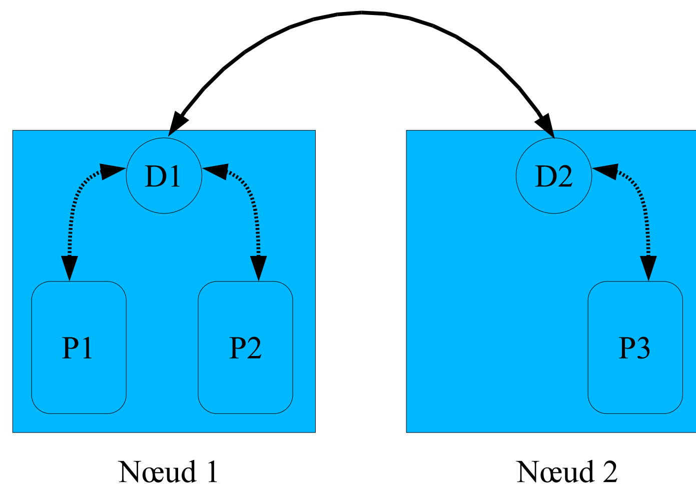


Fig. 2.1 – Transmission de messages dans PVM

Les efforts combinés de la communauté scientifique et du monde industriel ont amené à la réalisation d'une spécification commune à tous, devenue *de facto* un standard : MPI (*Message Passing Interface*)[30, 31]. Le standard MPI est distribué à la fois dans des versions fournies par les constructeurs de matériel et dans des versions gratuites, principalement LAM[21, 80] et MPICH[38]. Il existe des bibliothèques pour écrire des programmes MPI dans la plupart des langages utilisés dans le domaine de la haute performance (Fortran, C, C++...).

Architectures de type réparti Alors que les architectures parallèles ont pour objectif principal la performance, un second type d'architecture a été élaboré en privilégiant la répartition géographique des fils d'exécution. La norme CORBA (*Common Object Request Broker Architecture*[64]) définie par l'OMG (*Object Management Group*[3]) est une architecture répartie reposant sur un modèle d'objets. Lors de la conception d'une application, le problème à résoudre est décomposé en un ensemble d'objets. Les communications se font au travers de requêtes échangées entre les objets. Le transport et la remise de ces requêtes sont assurés par l'ORB (*Object Request*

Broker), indépendamment de la localisation physique des objets et ceci en toute transparence pour ces objets. Plusieurs intergiciels fournissent une implémentation de la norme CORBA : MICO[2], TAO[73] et OmniORB[9] en sont des exemples *OpenSource* issus des milieux académiques. Des interfaces CORBA existent pour la plupart des languages de programmation existants.

Architectures fondées sur un langage de programmation Le langage de programmation à objets JAVA, conçu par *Sun Microsystem*, dispose du mécanisme RMI (*Remote Method Invocation*) permettant à un objet d'invoquer une méthode d'un autre objet sans se soucier de la localisation de ce dernier.

La description de la requête de communication est incluse dans le langage lui-même, et c'est la machine virtuelle JAVA qui assure le transport et la remise de la requête. Dans une telle architecture, aucun intergiciel spécifique n'est nécessaire : le langage suffit pour faire communiquer les objets.

Ce sont ces environnements de communication que l'on retrouve principalement sur les grappes de calculateurs pour déployer des applications parallèles conçues sur le modèle d'échange de messages. Ces environnements offrent tous un modèle de programmation au travers :

- d'une abstraction des matériels de communication,
- d'une abstraction des nœuds de calcul ainsi que des mécanismes pour le déploiement,
- d'un mécanisme de surveillance des processus de l'application programmée.

Par ailleurs, tous ces environnements de communication délèguent l'exécution des communications à ce que nous appelons les primitives de communication.

Le chapitre 1 nous a permis d'énoncer plusieurs propriétés qui nous semblent fondamentales dans un système à image unique pour le support d'exécution d'applications parallèles communiquant par message. Ces propriétés sont :

- la performance offerte aux applications par le système de communication du système à image unique,
- le support au déplacement de processus communiquant par message (propriété 6),
- la tolérance aux fautes des applications communiquant par message (propriété 11).

De plus, il est essentiel que les performances des primitives de communication soient conservées, au niveau des applications, sans dégradation. En effet, dans le domaine de la haute performance, peu de compromis sont acceptables sur ce point.

Dans la suite de ce chapitre nous étudions différents travaux effectués sur ces trois propriétés dans le contexte des environnement de communication, l'accent étant mis principalement sur les environnements de communication évolués de type architecture parallèle.

2.3 Recherche de la performance dans les environnements de communication par message

La problématique de l'amélioration des performances d'une application est probablement aussi ancienne que le monde des machines à calculer. Lorsqu'il s'agit d'améliorer les performances d'une application conçue sur le modèle d'échange de messages, il est naturel de chercher à améliorer les performances du système de communication.

Lorsque l'application est conçue à partir des primitives de communication, un des seuls changements possible consiste à remplacer la primitive par une autre plus performante pour l'application. Selon que la primitive soit de type matériel ou de type protocole de communication, les modifications à apporter à l'application sont plus ou moins complexes.

De part sa conception, l'environnement de communication peut utiliser un grand nombre de primitives de communication différentes sans affecter l'application. Ainsi, pour l'environnement de communication MPICH, la primitive de communication par défaut est le protocole TCP/IP, et ceci quelque soit le matériel de communication réellement exploité. L'ajout d'un gestionnaire spécifique à un matériel ou une primitive existante, par exemple MPICH-GM[62], permet à MPICH d'utiliser les primitives de communication naturelles pour le matériel de communication employé (carte Myrinet et bibliothèque GM dans ce cas). Dans ce cas précis, l'application ciblée nécessite d'être compilée mais n'a pas besoin d'être modifiée.

Le projet Starfish[4] est un autre environnement de programmation MPI, orienté haute performance et tolérance aux fautes. Cet environnement sépare les communications pour la gestion du système (gérées à l'aide d'Ensemble[41]) des communications pour l'application. Afin que ces dernières soient les plus performantes possibles, Starfish emploie une interface de bas niveau (*i.e* BIP sur Myrinet) et évite autant que possible le recours à des appels systèmes (par exemple en employant l'attente active (avec faible niveau de priorité) en cas de réception).

Dans chaque cas, l'aspect haute performance du projet repose sur l'usage d'une primitive de communication elle-même haute performance.

2.4 Support au déplacement des processus dans les environnements de communication

Déplacer un processus entre deux nœuds consiste à :

1. stopper l'exécution du processus sur le nœud d'origine,
2. extraire un état correct du processus,
3. envoyer l'état du processus au nœud destinataire,
4. instancier le processus sur le nœud de destination et redémarrer son exécution à partir de l'état capturé.

Lors du déplacement d'un processus, cette succession d'étapes peut être réalisée au sein de l'application elle-même qui gère seule son déplacement. Dans la suite de ce

document, nous ne nous intéressons pas à ce type d'approche. Nous nous intéressons aux systèmes qui visent à offrir une transparence complète de la migration de processus vis-à-vis de l'application.

Des solutions moins spécifiques ont été étudiées par de nombreux projets. Migratable PVM (MPVM)[24] et UPVM[46] sont des extensions de PVM permettant le déplacement transparent de processus d'application PVM. Ainsi un processus faisant partie d'une application parallèle PVM peut-être stoppé sur un nœud et redémarré sur un autre nœud de manière transparente pour l'application. La transparence est obtenue par l'extension des mécanismes internes de PVM, sans en modifier les interfaces ainsi que par l'interception de certains appels systèmes. Seuls les binaires relatifs à l'environnement communiquant PVM sont modifiés, l'application reste inchangée. Du fait que PVM offre un modèle de machine parallèle, l'identification des processus est fournie par PVM. Les modifications concernant les communications se situent principalement dans les primitives pvm_send et pvm_recv. Ces fonctions sont modifiées pour localiser l'interlocuteur et communiquer avec le *daemon* associé.

AMPI[42] est un environnement MPI exploitant le système de communication issu de CHARM++[48]. AMPI est conçu autour d'un système de processus léger utilisant un *isomalloc*[6] pour les parties systèmes des processus légers (notamment pour la pile). Ainsi l'ensemble des processus MPI sont représentés par des processus légers. Le déplacement d'un processus MPI est donc réalisé entièrement par l'environnement AMPI.

Dans le système CoCheck[68], qui est une extension d'un environnement de communication par échange de messages mettant en œuvre des points de reprise, le déplacement de processus est réalisé par la sauvegarde et la restauration de points de reprise. Un point de reprise permet à une application de reprendre son exécution à partir d'un état sauvegardé et ainsi de ne pas perdre la totalité du temps de calcul déjà investi. Dans le cadre de la défaillance d'un nœud dans une grappe de calculateurs, l'application doit pouvoir être redémarrée à partir d'un ensemble différent de nœuds. C'est à l'aide de ce mécanisme que CoCheck effectue le déplacement de processus communicants (en l'absence de défaillance). Cocheck réalisant les points de reprise de manière transparente pour l'application, il se pose le problème de l'adressage des processus. En effet, le déplacement étant transparent, les processus doivent conserver leur identifiant malgré ce déplacement. CoCheck fournit des adresses virtuelles aux processus pour leur identification, et établit de manière interne une table de correspondance avec les adresses physiques réelles. Grâce à cette indirection dans l'adressage des processus, la transparence est respectée.

Enfin, Mosix[14] et OpenSSI[40], qui sont des systèmes à image unique, résolvent le problème de déplacement de processus communicants par une indirection des communications par le nœud de création du processus. Ainsi, la communication initiale n'est pas modifiée mais un relai est établi entre l'extrémité initiale de la communication et la localisation courante du processus.

2.4.1 Tolérance aux fautes dans les environnement de communication par des techniques de point de reprise

Une technique courament employée à des fins de tolérance aux fautes, dans le domaine des applications scientifiques à haute performance, est le recouvrement arrière : une application sauvegarde périodiquement son état en mémoire stable. Les sauvegarde sont appelées points de reprise. En cas de défaillance, l'application peut être redémarrée à partir d'un point de reprise. Cette technique est généralement associée à un modèle de faute de type *arrêt sur faute*. La machine rejouant l'application n'étant pas obligatoirement celle ayant calculé le point de reprise, cet outil peut-être employé pour effectuer des déplacements de processus. Ainsi, de nombreux systèmes que nous détaillons dans la suite de ce paragraphe apportent également une solution pour la migration de processus communicants.

Capture de l'état d'un processus Afin de créer un point de reprise pour un processus, il est nécessaire de capturer l'état complet de ce processus. L'état d'un processus se compose de deux parties distinctes : la mémoire (segment de code, données et pile) associée au processus, et les états du système d'exploitation (registres du processeur, état des fichiers ouverts, etc.) associés à ce processus.

La bibliothèque *libckpt*[68] permet de réaliser des points de reprise pour un processus Unix non communicant. Libckpt permet la création de points de reprise de manière automatique ou à la demande de l'application elle-même. Les points de reprise sont incrémentaux et enregistrés dans un fichier. Hormis la contrainte sur le type de processus, il est nécessaire de modifier une ligne du code source de l'application (remplacer la déclaration main() par checkpoint_target) et d'effectuer l'édition de lien avec une bibliothèque statique (*libckpt.a*). Une plus grande transparence est obtenue par le système Condor[52] qui allie la transparence de son architecture à un mécanisme de point de reprise de processus non communicant.

Capture de l'état de processus communiquant par échange de messages Concernant les processus communiquant par échange de messages, la création d'un point de reprise est plus délicate. Une communication par message implique deux processus distincts (appelés A et B). L'état d'un tel ensemble est composé de :
- l'état du processus A,
- l'état du processus B,
- l'état du canal de communication.

Le canal de communication peut contenir un ou plusieurs messages émis et acquittés du point de vue de l'émetteur, mais non encore délivrés du point de vue du récepteur. La situation est pire si nous considérons qu'un long message peut être en cours d'émission au moment de la demande de capture de l'état du canal. L'état d'un canal de communication est donc réparti entre les deux processus communicants, les deux systèmes d'exploitation concernés, et le matériel de communication qui a un moment donné peut détenir un fragment du message. Cet aspect distribué du point de reprise fait qu'une grande partie des travaux concernant les points de reprise d'applications communiquant

par échanges de messages concerne les environnements de communication de type PVM ou MPI.

À notre connaissance, il n'existe pas d'outils semblables à *libckpt* permettant d'apporter de manière simple un mécanisme de point de reprise à un processus quelconque communiquant par échange de messages. Cependant, il existe pour des environnements de communication ciblés, tels que MPI, des systèmes permettant d'apporter la propriété de recouvrement arrière. Dans la suite de ce paragraphe, nous présentons deux approches pour la réalisation de point de reprise au sein d'environnement de communication : les points de reprise par synchronisation globale ainsi que ceux par journalisation des événements de communication.

Les techniques pour capturer l'état des processus A et B sont similaires à celles employées précédemment par *libckpt*. La difficulté concerne la capture de l'état du canal de communication. Une première approche (tuMPI basée sur CoCheck[81], Starfish[4], Fail-Safe PVM[50]) consiste à s'assurer que le canal de communication est vide. Cette technique consiste pour un processus (disons A) en l'émission d'un message particulier sur l'ensemble des canaux de communication avant de créer un point de reprise. Lorsqu'un processus quelconque (disons B) dans la grappe reçoit un tel message, il s'assure de ne plus rien avoir en cours d'émission sur le canal, puis retourne le message de contrôle. Au retour du message de contrôle (et en supposant que les messages soient traités dans l'ordre où ils sont émis) le canal de communication est considéré vide : il n'y a plus de message en cours de transfert. L'état du canal est donc déterminé et peut être capturé. Ultérieurement, la restoration du canal impose une restoration de ses deux extrémités. En effet, si après la création du point de reprise de A, B envoie un message, et A subit une défaillance, alors au redémarrage de A, le message est perdu car A ne l'aura pas reçu et B ne le re-émettra pas.

Par conséquent pour être correcte, cette technique de point de reprise d'un canal de communication, nécessite que la restoration se fasse simultanément sur l'ensemble de ses extrémités. Le calcul d'un point de reprise global à l'application parallèle (et non pas simplement à un processus en particulier) est donc nécessaire. Un point de reprise global correspond à la création coordonnée des points de reprise de chacun des processus de l'application.

Comme nous venons de le voir, "vider" les canaux de communication impose une forme de synchronisation globale de l'application. Cette opération étant coûteuse, augmenter la fréquence des points de reprise ou augmenter le nombre de processus dans l'application parallèle peut, au final, être jugé rédhibitoire. Une autre approche consiste donc à permettre la création de points de reprise non-coordonnés. Lors du re-démarrage d'un processus à partir d'un tel point de reprise, les communications sont rejouées exactement de la même manière que s'il n'y avait pas eu d'interruption. En particulier, l'ordre et le contenu des messages sont préservés. C'est cette approche que des projets tels que MPICH-V[20, 19, 75] et MPI/FT[16] ont retenu. Contrairement à l'approche précédente où les canaux de communication sont ramenés à un état prédéfini (c'est-à-dire vide), le problème est de pouvoir capturer l'historique complet d'un canal de communication. À l'aide d'un tel historique, l'application re-démarrée peut recevoir les messages qui lui sont destinés de manière complètement transparente pour l'émetteur.

De même les messages émis en doublon par le processus redémarré sont détruits par le canal de communication sans créer d'interférence avec l'interlocuteur. Ainsi, grâce à l'historique, le processus re-démarré peut être remis à niveau par rapport aux autres processus de l'application parallèle, de manière complètement transparente pour ceux-ci. En pratique, MPICH-V et MPI/FT utilisent des nœuds intermédiaires pour gérer les canaux de communication (figure 2.2). Ces intermédiaires (appelés *Channel Manager* chez MPICH-V ou *Coordinator* chez MPI/FT) enregistrent tous les messages transmis et peuvent simuler les événements en cas de redémarrage d'un processus.

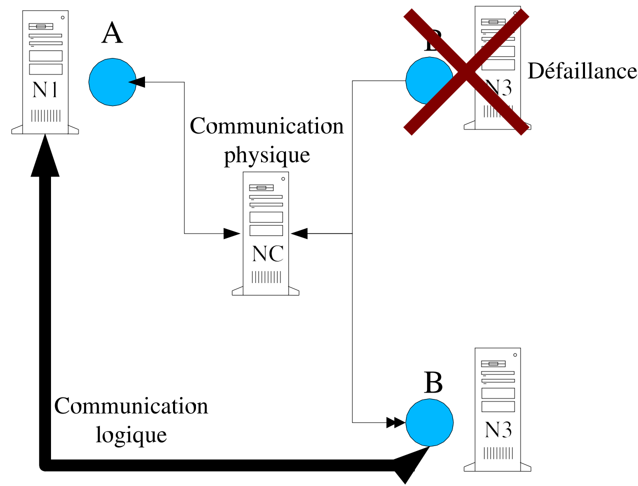


FIG. 2.2 – Communication à l'aide d'intermédiaire pour la tolérance aux fautes

2.5 Conclusion

Dans ce chapitre, nous nous sommes placés sous l'angle des problèmes de communication liés aux modèles de programmation possibles sur une grappe de calculateurs. Hormis le modèle hybride, ces modèles de programmation sont la programmation par mémoire partagée et la programmation par échange de messages. L'utilisateur de ces modèles souhaite bénéficier de propriétés telles que la haute performance de son application, la possibilité de déplacer un processus de son application, ou encore de bénéficier de mécanisme de tolérance aux fautes tel que la création de points de reprise.

Les applications conçues sur le modèle du partage de mémoire dépendent d'un support matériel ou logiciel pour réaliser le partage de la mémoire. Dans les deux cas, les échanges de données sur le réseau d'interconnexion sont masqués par le système de partage de la mémoire. Dans le cas des grappes de calculateurs, le partage mémoire est mis en œuvre par un service distribué. Ce sont donc les performances des communications du service distribué qui vont influencer les performances de l'application.

Concernant les applications conçues selon le modèle d'échange de messages, les communications sont explicites pour l'application.

Le déplacement d'un processus nécessite un support de la part du système de communication afin de transmettre les messages vers la localisation courante d'un processus et non pas vers la localisation du processus lors de sa création de la communication. Deux classes d'approches existent face au problème de la localisation. Une première solution consiste à employer une indirection fixe (par exemple, la localisation initiale du processus). Cette indirection a pour but de suivre les déplacements du processus communiquant et ainsi de relayer les messages vers la destination courante du processus. La seconde approche consiste à utiliser, de manière interne au système de communication, une table d'indirection pour l'adressage des processus.

Le problème de la tolérance aux fautes d'applications communiquant par échange de messages est lié à celui de la capture de l'état des canaux de communication. Afin de capturer cet état, il est possible de ramener le canal à un état pré-déterminé (*ie.* vide de tout message) ou d'enregistrer tous les événements du canal de manière à les rejouer ultérieurement. Lors d'une faute, cette dernière technique permet de ne redémarrer que le (ou les) processus en cause, sans perte du travail réalisé par les processus corrects. Dans ce cas, le processus est redémarré et le canal de communication rejoue, à partir de son propre historique, les événements de communication de manière à re-synchroniser le processus avec le reste de l'application.

Chapitre 3

Infrastructure de communication dans les grappes de calculateurs

De part son architecture distribuée, les performances globales d'une grappe de calculateurs sont fortement liées à celles de la technologie d'interconnexion des nœuds. Les caractéristiques de ce système d'interconnexion sont à la fois déterminées par le modèle de communication retenu, les caractéristiques de la technologie de communication choisie, ainsi que les propriétés du système de communication.

Dans la suite de ce chapitre, nous détaillons les différents types de technologie de communication communément rencontrés dans les grappes de calculateurs, ainsi que les différents modèles de communication développés sur ces technologies. Enfin nous présentons plusieurs environnements pour des applications communiquant par message, en gardant à l'esprit les caractéristiques propres aux systèmes à image unique dégagées au chapitre 1.

3.1 Infrastructure matérielle et gestionnaires de communication

Sans aucune modification, un système d'exploitation de type Linux offre un environnement logiciel pour les communications entre les nœuds d'une grappe. L'approche traditionnelle place le système d'exploitation au cœur de toute communication puisque chaque émission/réception se fait par l'intermédiaire d'appels système. Un appel système correspond à un point d'entrée du SE permettant aux applications d'exécuter du code privilégié. Certains systèmes de communication exploitent cette caractéristique en imposant des appels système pour toutes les communications. Cependant, pour des raisons liées à la sécurité du SE (mécanisme d'appel système, changement de contexte application/SE, etc.), un appel système se révèle plus coûteux qu'un simple appel de fonction. Ce constat a amené d'autres projets de recherche à essayer de limiter, autant que possible, les interactions avec le système d'exploitation.

La suite de ce paragraphe porte sur une présentation de ces deux approches dans la conception d'un système de communication.

3.1.1 Système de communication par le SE

Il existe de très nombreuses technologies d'interconnexion dans le monde des SAN et des LAN. La technologie la plus répandue est très certainement le réseau Ethernet[78]. Cette technologie, apparue en 1975, crée virtuellement un bus entre les différents participants au réseau. Initialement définie avec une bande passante de 10Mbits, la norme Ethernet a, par la suite, été déclinée en FastEthernet (100Mbits), GigabitEthernet (1000Mbits) et maintenant le 10 GigabitsEthernet. Sur un réseau Ethernet, les données circulent dans des trames. Les trames ont une taille limitée et incluent à la fois un en-tête et le corps du message. La destination de la trame est encodée dans son en-tête sous la forme de l'adresse MAC de l'interface de destination. Le système d'exploitation assiste le matériel dans toutes les phases de la communication.

La figure 3.1 présente schématiquement les manipulations d'un message réalisées par le SE Linux (version 2.4) lors d'une transmission.

L'infrastructure réseau du SE Linux est construite sur le modèle Ethernet. Lorsqu'un processus demande l'émission d'un message, celui-ci est préparé sous forme de trames directement manipulables par le gestionnaire de périphériques contrôlant le matériel de communication. Le transfert de données entre le SE et le matériel se faisant par un DMA (*Direct Memory Access*), les données doivent respecter plusieurs propriétés liées à la mémoire. Notamment, ces données doivent être présentes dans une mémoire physique adressable par le contrôleur de DMA. De plus, des contraintes de contiguïté des données doivent être respectées. Pour ces raisons, le SE peut réaliser des traitements supplémentaires lors de la création des trames, tels qu'une recopie des données. Pour ces traitements, le SE fait usage de zones mémoire de travail que nous appellons **tampon**.

Définition 21 ***Tampon de communication****: Nous appelons* ***tampon de communication*** *un espace mémoire interne au système de communication, qui a pour but de contenir tout ou partie d'un message en vue d'un traitement interne au système de communication.*

Dans le SE Linux, les tampons de communication correspondent aux structures **struct** sk_buff.

Une fois les trames créées, elles sont transmises à un moteur d'émission qui contrôle l'alimentation du matériel de communication. En retour, ce matériel informe le SE de l'émission d'une trame par une interruption. Lorsqu'une trame est émise, le moteur d'émission peut en fournir une nouvelle au matériel.

Du côté du récepteur (figure 3.2), le signal de réception d'une nouvelle trame est lui aussi porté par une interruption. Cette interruption permet le déclenchement du traitement de la trame par le moteur de réception lié au protocole, qui effectue éventuellement une recopie des données à sa destination finale. Finalement, selon le type de protocole, le moteur de réception informe le processus récepteur (par exemple en le réveillant) de l'arrivée d'un message. Le temps de transfert d'un message d'un processus émetteur vers un processus récepteur correspond donc à la somme des temps :

– de préparation des trames,

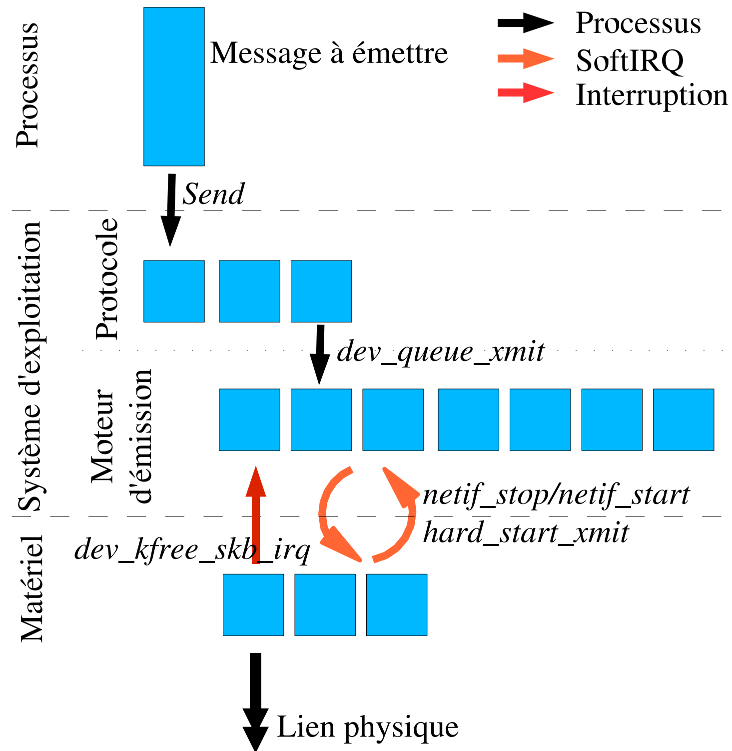


Fig. 3.1 – Pile réseau d'un noyau Linux, à l'émission

- d'attente avant transfert des trames,
- de transfert des trames,
- de réception des trames,
- d'analyse des trames par le protocole dédié,
- de signal (réveil) du processus récepteur.

Dans le cas des réseaux Ethernet et FastEthernet, les machines modernes peuvent *pipeliner* le traitement de plusieurs trames, c'est-à-dire effectuer le traitement d'une trame différente à chaque étape de transmission. Dans ce cas, le coût du transfert d'une trame domine celui de sa préparation et de sa réception. Grâce au *pipeline* le débit maximal théorique est atteint. Toutefois, avec les évolutions récentes d'Ethernet, le coût de transfert n'est plus l'élément prépondérant, et le débit théorique n'est plus atteint avec les machines standard. En effet, les recopies ainsi que le temps de signal

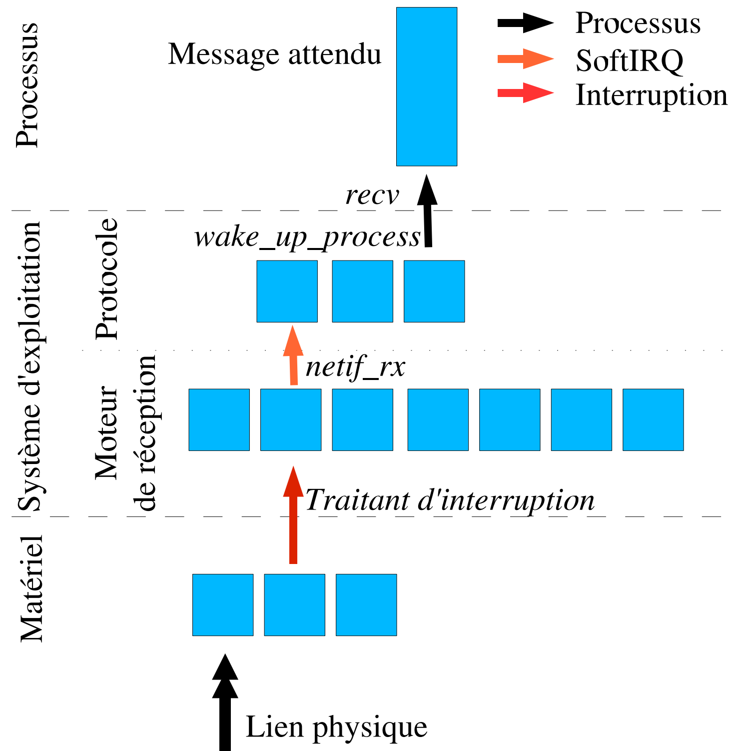


Fig. 3.2 – Pile réseau d'un noyau Linux, à la réception

du processus récepteur deviennent les éléments prépondérants.

En particulier, lorsque le signal est représenté par le réveil d'un processus bloqué en réception, seul l'ordonnanceur détermine le prochain accès du processus à la ressource processeur. En résumé, bien qu'un événement (une interruption) soit immédiatement déclenché à la réception d'une trame, et que le protocole associé soit exécuté au plus tôt (par un SoftIRQ), la réception effective par le processus ne peut pas être précisement déterminée dans le cas général.

Ainsi le processus communicant n'accède pas directement à l'interface de communication mais passe par le système d'exploitation. Bien que cette approche permette un partage transparent de la ressource de communication ainsi qu'une vision générique des interfaces de communication (pourvu que celles-ci soient supportées par le système d'exploitation), la contre-partie est de payer le coût du changement de contexte entre

l'espace utilisateur et l'espace noyau. De plus, une recopie des zones mémoires est souvent nécessaire. En effet, les échanges avec les interfaces de communication imposent bien souvent l'usage d'un ensemble de pages mémoire contiguës physiquement, propriété qui ne peut être garantie par l'application. Afin d'éviter le surcoût du changement de contexte, le projet *Genoa Active Message MAchine* (GAMMA)[26, 27] propose le remplacement de la pile de communication classique (avec les appels système) par une pile optimisée pour les performances en réseau local.

3.1.2 Accès direct aux matériels de communication

Lorsque le problème de la réduction des coûts de communication a été étudié, le passage obligé par le système d'exploitation a été identifié comme grandement pénalisant. Des projets tels que *U-Net*[97, 99] ont proposé des systèmes de communication supprimant le système d'exploitation du chemin critique de l'émission et de la réception de messages. Ainsi, le surcoût lié aux appels systèmes est-il supprimé et la gestion des tampons de communication, à l'instar de ceux de l'application, peut être réalisée en espace utilisateur. La partie A de la figure 3.3 décrit l'usage traditionnel du SE lors des communications : toutes les opérations se font par son intermédiaire. Á l'opposé, *U-Net* (partie B de la figure) limite les appels système aux phases d'initialisation et de finalisation des canaux de communication. Une fois le lien établi, l'application réalise ses communications sans recours côuteux au SE. Concernant les tampons de communication, ceux-ci font partie de l'application et sont fournis par le système de communication lui-même. Cette approche consiste en une première étape vers un système de communication sans recopie (autre que vers et en provenance du matériel de communication).

Myrinet[18] de la société Myricom, est un exemple de système de communication matériel haute performance permettant des communications sans recours systèmatique au SE. La technologie Myrinet est fondée sur l'usage d'un processeur embarqué dans l'interface de communication pouvant prendre en charge une partie des opérations de communication sans intervention d'un processeur de la machine hôte. Ce processeur dispose de sa propre mémoire et peut provoquer des échanges de mémoire avec la mémoire principale de la machine hôte. Myricom livre avec sa technologie Myrinet une bibliothèque de communication appelée GM. Ce logiciel se décompose en trois parties intimement liées :

- le *firmware* : logiciel exécuté par le processeur embarqué,
- un gestionnaire de périphérique, permettant au SE de contrôler l'interface de communication,
- une bibliothèque fournissant l'ensemble de l'interface de programmation de GM.

Seule cette bibliothèque est directement accessible aux applications souhaitant exploiter GM sur Myrinet. GM fournit un mécanisme d'adressage au sein d'un réseau Myrinet. Les liens de communication sont instanciés au travers de la fonction `gm_open()`. Cette fonction réalise au travers d'un appel système l'initialisation d'un nouveau lien sur l'interface de communication et fournit à l'application un descripteur de communication. Typiquement, cet appel système n'est réalisé qu'une fois, au moment du

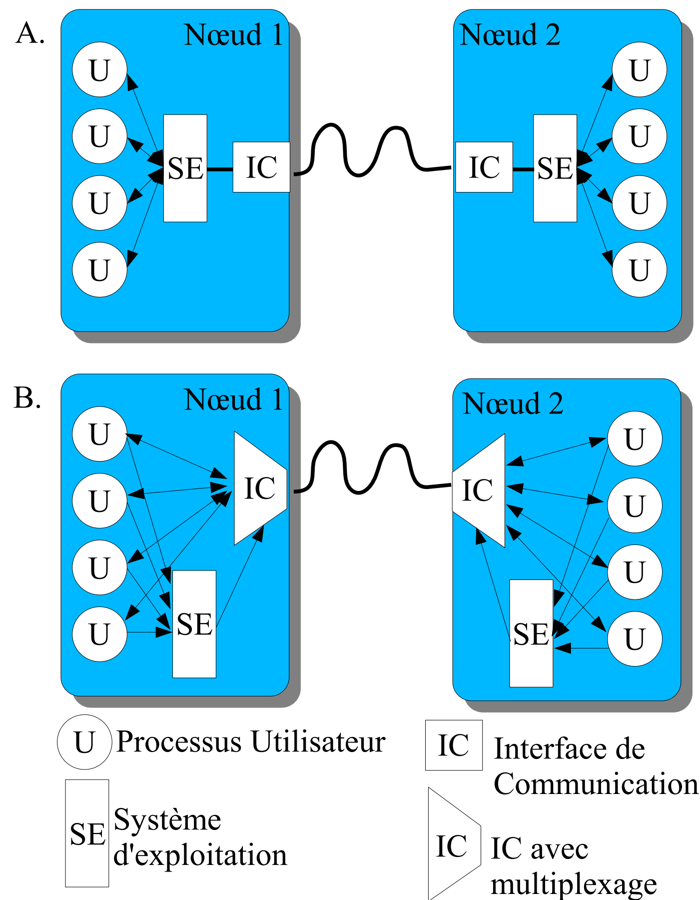


FIG. 3.3 – U-Net : limitation du recours au système d'exploitation

déploiement de l'application. Les tampons de communication sont obtenus du système de communication par l'intermédiaire de fonctions du type gm_dma_malloc. À nouveau par un appel système, de la mémoire est réservée pour l'application et déclarée auprès du système de communication (et en particulier du processeur embarqué) pour un usage ultérieur. Techniquement parlant, cette mémoire est marquée par le SE de manière à n'être déplacée en aucun cas. Ainsi, l'adresse mémoire physique correspondant à ce tampon peut être conservée par le processeur embarqué sans crainte de changement inopiné. Munie du descripteur ainsi que des tampons de communication, l'application peut poster des requêtes de communication à l'aide des primitives gm_send_with_callback et gm_receive, gm_receive_pending. Ces requêtes décrivent le type de communication (émission ou réception) ainsi que les zones mémoires concernées par la requête. Le reste de l'opération est intégralement pris en charge par l'interface de communication sans aide

supplémentaire du système d'exploitation. Par ailleurs, ce modèle de communication impose que le système de communication (GM) dispose en permanence d'un ensemble de tampons de communication en nombre et taille suffisants. Dans le cas contraire, le système de communication ne réalise pas d'allocation de tampon supplémentaire et le comportement peut-être indécis. Par conséquent, la véritable contrainte pour une application utilisant GM est de réaliser une gestion correcte de ses tampons de communication et de les restituer en temps voulu à GM par l'intermédiaire de fonctions comme gm_provide_receive_buffer.

Dans le contexte spécifique des grappes de calculateurs, il est possible de faire des hypothèses encore plus fortes concernant la programmation des communications : c'est le sens du projet BIP[94, 70, 71]. BIP est un système de communication complet (*firmware* et gestionnaire de périphérique) pour les interfaces Myrinet remplaçant complètement GM.

BIP impose que les grands messages soient préalablement attendus et surtout que les petits messages soient traités suffisamment rapidement pour ne pas être écrasés dans les tampons de réception du système de communication. Dans le cas où cette dernière hypothèse ne serait pas vérifiée, les pertes de messages silencieuses sont possibles. Ces hypothèses extrêmes permettent à BIP de s'affranchir de plusieurs mécanismes considérés comme coûteux (par exemple le contrôle de flux pour les petits messages).

Dans les deux technologies précédentes, l'unité de communication est le message. Celui-ci est éventuellement encapsulé dans une requête et traité par l'interface de communication. Une autre approche consiste à considérer la page mémoire comme étant l'unité de communication. La technologie SCI[34] de Dolphin permet le partage transparent de plusieurs pages mémoire entre plusieurs nœuds différents. Une telle technologie exploite les mécanismes de mémoire virtuelle du système d'exploitation afin de projeter dans l'espace d'adressage d'un processus une portion de la mémoire embarquée sur l'interface de communication.

En dépit des différences technologiques, l'ensemble des interfaces de communication réseau précédemment évoquées dispose d'une adresse unique permettant de l'identifier. Cet identifiant offre un service de désignation matériel. Ce système de désignation permet d'émettre des données à destination d'une autre interface de communication (déterminée par son identifiant unique).

3.2 Modèles de communication

Les modèles de communication définissent le fonctionnement interne des applications communicantes. De tels modèles peuvent directement découler de l'environnement logiciel (gestionnaire ou bibliothèque de bas niveau) associé aux matériels de communication, ou au contraire permettre une forme d'abstraction des technologies employées. Cette abstraction est définie à partir des besoins de l'application et non

pas des spécifications du matériel. Ces abstractions correspondent aux primitives de communication évoquées dans le paragraphe 2.2.1.

Dans la suite de cette partie, nous étudions, sous l'angle des performances, différents modèles de communication.

Modèle de communication synchrone Lors d'une communication, deux rôles sont à définir : l'émetteur et le récepteur. Bien que fréquemment symétriques, ces rôles doivent être étudiés séparément.

Le processus émetteur prépare son message dans un tampon de l'application, transmet le tampon au système de communication et attend la fin du traitement par celui-ci. Cette fin de traitement est en théorie représentée par la réception d'un acquittement du nœud de destination indiquant que le message est correctement réceptionné. Selon la qualité du réseau d'interconnexion, une ou plusieurs retransmissions peuvent s'avérer nécessaires et donc pénaliser le processus émetteur. Une optimisation simple consiste, pour le système de communication, à effectuer une recopie du message dans un de ses tampons, de manière à libérer le processus émetteur le plus rapidement possible. Symétriquement, le processus récepteur fournit au système de communication un de ses tampons, et attend que le système de communication remplisse le tampon.

Un tel modèle est appelé **modèle de communication synchrone**. Un processus conçu autour d'un tel modèle doit :

- mettre en forme le contenu de ses messages à émettre,
- à chaque requête de communication, interrompre son exécution au profit du système de communication,
- analyser le contenu des messages reçus.

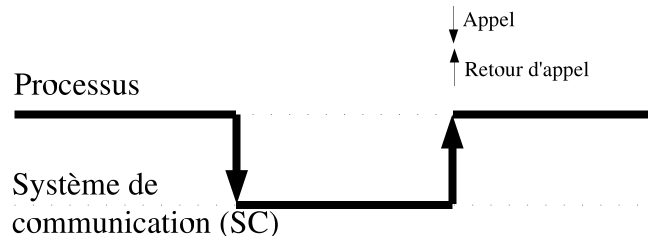


Fig. 3.4 – Chronogramme d'une communication synchrone

Les interfaces les plus répandues pour ce type de communication sont les pipe, les sockets BSD[83] et les streams[84].

Dans le contexte d'un processus attendant des messages de provenances différentes, et devant appliquer un traitement spécifique à chacun de ces messages, un modèle de communication synchrone peut s'avérer pénalisant. En effet, quelque soit l'ordre d'arrivée réel des messages, ils sont traités par le processus dans l'ordre choisi par le programme. Un tel modèle de communication impose à un processus d'attendre un message au lieu de traiter ceux déjà arrivés. Dans le modèle synchrone, le processus récepteur peut résoudre ce problème en exploitant une fonction de multiplexage des

opérations synchrones (correspondant à l'appel système select ou poll des systèmes Unix). Le processus demandé est mis en attente d'un événement quelconque sur un ensemble de canaux. À chaque occurrence d'un événement, le processus peut effectuer la demande de réception correspondante. Ainsi le message est traité et le processus peut de nouveau se placer dans l'attente d'un événement.

Modèle de communication asynchrone Par conception, un processus exploitant une fonction de multiplexage synchrone ne peut effectuer la réception qu'après la réception physique du message par la machine. Une recopie du message est donc imposée par le système de communication afin de transférer le message du tampon utilisé par le système de communication à celui fourni par le processus récepteur.

À l'émission comme à la réception, le modèle synchrone spécifie implicitement le moment où le processus peut ré-utiliser les tampons de communication :

– à l'émission : un nouveau message peut être construit dans le tampon,
– à la réception : le message attendu est effectivement présent dans le tampon.

Toutefois, selon les caractéristiques du message, de telles recopies peuvent nuire aux performances globales du système de communication. De plus la nature même du modèle de communication synchrone ne permet aucune forme de parallélisme entre l'application communicante et le système de communication.

Le **modèle de communication asynchrone** permet à un processus de poster une requête de communication (en émission ou en réception) associée à un tampon du processus et de poursuivre son exécution. Ultérieurement, le système de communication va acquitter la requête du processus. Ce modèle spécifie que le processus ne doit pas modifier son tampon jusqu'à l'acquittement du système de communication.

Ainsi, dans le cas de machines à plusieurs processeurs, ou exploitant une interface de communication dotée de son propre processeur, le processus peut s'exécuter en parallèle de l'envoi du message. Le modèle de communication asynchrone permet d'anticiper les communications et ainsi de recouvrir le temps de communication par du temps de calcul.

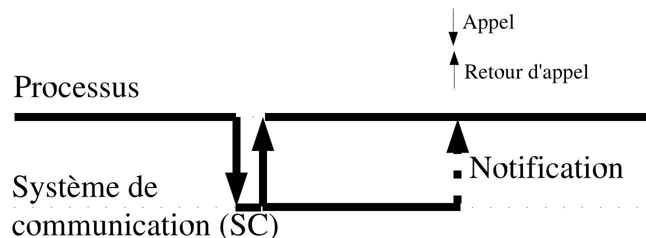


Fig. 3.5 – Chronogramme d'une communication asynchrone

L'interface de communication des *sockets* autorise un mode asynchrone. Dans ce mode, l'ensemble des opérations sont non bloquantes et celles-ci sont acquittées par l'envoi du signal SIGIO au processus communicant.

Modèle de communication par composition de message Dans les deux modèles précédemment présentés, les messages doivent être construits dans le tampon qui servira à la requête de communication. En particulier, si le message est un assemblage de différentes données, celles-ci doivent d'abord être agrégées dans le tampon avant d'être transmises au système de communication. Le problème est analogue pour le récepteur qui pourrait souhaiter recevoir le message dans plusieurs zones mémoire distinctes. La raison pour laquelle un processus crée un tel message est de limiter le surcoût lié à l'envoi de nombreux petits messages. Le système de communication Madeleine[11, 10] offre une solution élégante à ce problème.

Madeleine est un environnement de communication haute performance fondé notamment sur un découpage fin des étapes de communication entre deux nœuds. Madeleine repose sur l'idée que seul le système de communication est à même de décider de la meilleure politique de communication, en fonction des contraintes des primitives de communication sous-jacentes et des exigences du processus client. Ainsi, pour l'émetteur, Madeleine propose une interface permettant la description du contenu d'un message (figure 3.1).

Listing 3.1 – Interface de communication send/receive

```
mad_begin_packing()  /* éDbut d'une transmission */
mad_pack()           /* Envoi d'uné élment */
mad_end_packing()    /* Fin d'une transmission */

mad_begin_unpacking()  /* éDbut d'une érception */
mad_unpack()           /* éRception d'uné élment */
mad_end_unpacking()    /* Fin d'une érception */
```

La figure 3.6 représente le schéma classique d'une émission à l'aide de Madeleine. Dans un premier temps, le processus client déclare une nouvelle communication, et décrit le premier fragment de message. La suite de la description du message se fait en parallèle de l'envoi effectif de celui-ci par le SC.

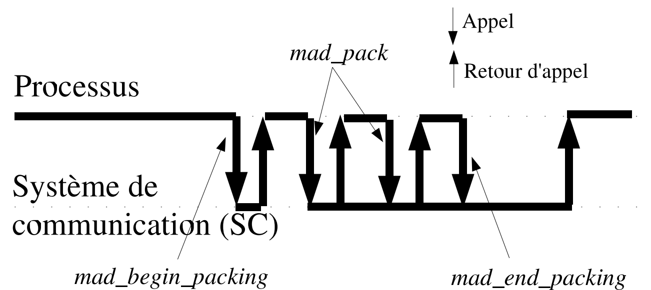


Fig. 3.6 – Chronogramme d'une communication avec "Madeleine"

À chaque élément ajouté dans le message est associée une contrainte de communication (figure 3.2). Ces contraintes concernent l'usage des tampons soumis au SC. Elles visent notamment à indiquer au SC si le tampon doit être traité immédiatement (mad_send_SAFER, mas_receive_EXPRESS) ou au contraire à un moment choisi par celui-ci (mad_send_CHEAPER, mad_receive_CHEAPER). La primitive signalant la fin de la requête est

bloquante. Grâce à ces différentes contraintes, le contrat concernant l'acheminement des données, entre le système de communication et son client, peut être modulé en fonction des besoins réels.

Listing 3.2 – Interface de communication send/receive

```
mad_send_CHEAPER /* \'Emission selon le SC*/
mad_send_LATER   /* éRserve un emplacementà  lé'mission , mais ne le fait pasé
                    immdiatement */
mad_send_SAFER   /* èProtge contre une modification éultrieure du tampon */

mad_receive_CHEAPER /* éRception selon le SC */
mad_receive_EXPRESS /* éRception bloquante */
```

Nous appelons **modèle de communication par composition de message**, un modèle de communication permettant d'émettre ou de recevoir un message sans utiliser un tampon intermédiaire uniquement employé pour le système de communication lui-même.

Modèle d'invocation de fonctions distantes Dans le modèle de programmation client/serveur, le processus récepteur est en attente permanente de messages pour déclencher une action. Ce type de processus permet à une machine d'offrir un service (et donc une partie de ses ressources) à d'autres machines.

Définition 22 ***Appel de fonction à distance****: Une machine peut enregistrer une liste de fonctions localement. Un* ***appel de fonction à distance*** *symbolise le fait qu'un autre nœud puisse demander l'exécution d'une telle fonction en fournissant des paramètres spécifiques et en attendant le résultat de cet appel.*

L'administration et la gestion d'un ensemble de machines au sein d'un réseau de stations de travail reposent sur la centralisation de certains services. Ces services sont rendus par certains nœuds spécialisés pour l'ensemble des autres nœuds. Ces services sont conçus sur le modèle de fonctions appelables à distance.

Appels de procédures à distance L'appel de procédures à distance (*Remote Procedure Call* – RPC)[17] est un mécanisme classique pour la communication entre processus dans un système réparti. Chaque service s'enregistre auprès d'un annuaire et se place en attente de requêtes en provenance d'autres parties du système réparti. Un exemple parmi les plus célèbres de ce type de service est certainement le système de fichier *NFS (Network File System)*[91] (*Network File System*) conçu par Sun Microsystems. Dans le cas d'ensembles de machines quelconques, et grâce au standard XDR (*eXternal Data Representation*), le protocole RPC tel que développé par Sun peut s'affranchir des problèmes d'hétérogénéité inhérents au système réparti. Ainsi les messages sont transmis sous la forme d'une représentation indépendante des matériels utilisés.

La figure 3.7 décrit, en terme d'usage du processeur, le mécanisme de réception et de traitement d'un message de type RPC. La première phase correspond à la réception

physique des trames par le matériel de communication. Ces trames sont traitées immédiatement dans le cadre d'une interruption liée au matériel. Ces trames sont assemblées en messages par le système de communication dès la fin de l'interruption matérielle (SoftIRQ). Le SC identifie le processus destinataire du message et l'en informe (généralement en débloquant ce processus). Dans le cas général, le traitement effectif du RPC doit attendre que l'ordonnanceur attribue la ressource processeur au processus concerné : un délai est ainsi créé.

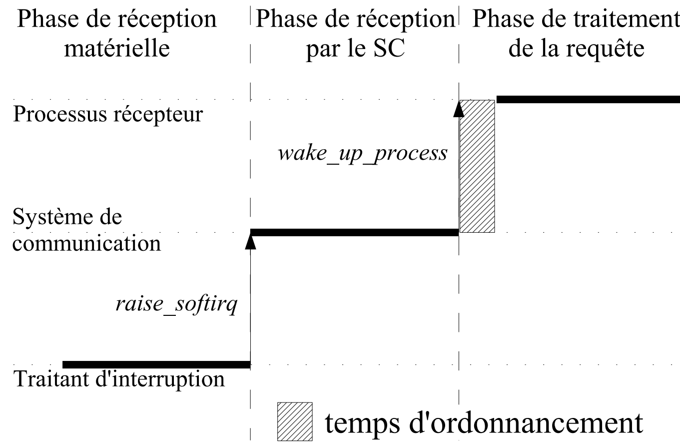


FIG. 3.7 – Chronogramme de la réception et du traitement d'un RPC

Message Actif Lors de la conception de services répartis, certaines requêtes se caractérisent par une action serveur très simple. Cette action peut être la mise à jour d'une valeur, la réalisation d'une opération binaire, ou l'envoi d'une donnée. Pour ce type d'opération très simple, le coût du traitement du message ainsi que celui du réveil du processus serveur peut largement dépasser celui de l'opération demandée. De plus, si le processus client est en attente d'une réponse du serveur, les problèmes de latence revêtent une grande importance pour les performances globales de l'application. Le projet *Active Message*[98] propose un modèle de communication particulier pour répondre à ce problème. En effet, les services peuvent enregistrer des fonctions devant être exécutées directement dans le contexte de la réception d'une trame par le gestionnaire de matériel de communication. Ainsi, chaque trame émise contient l'adresse de la fonction à exécuter. Ce modèle de communication présente le double avantage d'offrir une latence faible et de permettre à la requête d'être traitée indépendamment de la charge en processus du serveur. En effet, le traitement en interruption est, comme son nom le suggère, prioritaire par rapport aux processus du système.

La figure 3.8 représente, en terme d'usage du processeur, le mécanisme de réception et de traitement d'un message de type *Active Message*. Ce chronogramme est à mettre en correspondance avec celui des RPC (voir figure 3.7). La principale différence se situe au niveau du traitement du message. En effet, celui-ci est réalisé directement dans le cadre du SC sans problème de délai lié à l'ordonnanceur.

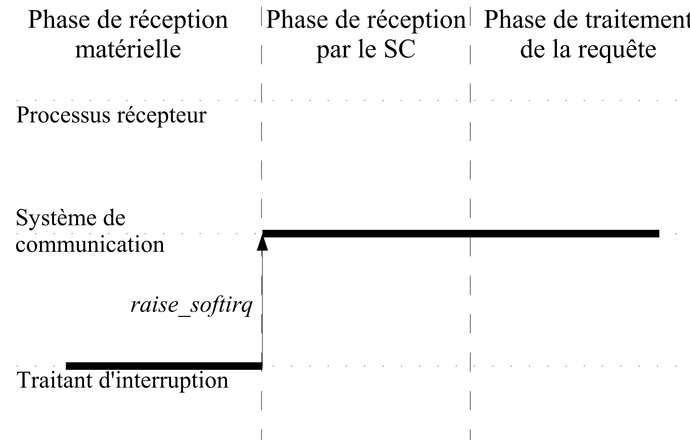


FIG. 3.8 – Chronogramme de la réception et du traitement d'un *Active Message*

3.3 Conclusion

Lors de la conception d'une application communiquant par échange de messages, un des premiers choix du développeur est celui du modèle de communication. Ces modèles se différencient par la sémantique associée aux messages, ainsi que par l'impact du SC sur l'exécution du processus communicant. Le tableau 3.1 synthétise les modèles présentés dans ce chapitre.

Modèles	Synchrone	Asynchrone	Composition de message	Invocation de fonctions distantes
Type de message	simple	simple	composé	simple
Type d'exécution	synchrone	asynchrone	synchrone	synchrone/asynchrone

TAB. 3.1 – Modèles de communication

La sémantique du modèle de composition de message permet de créer un lien logique entre des messages physiquement distincts sur le réseau d'interconnexion. Chacun de ces messages constitue un fragment du message logique. Une telle sémantique permet à deux messages composés d'être émis simultanément sur le même canal sans qu'ils ne se retrouvent mélangés.

Un facteur de performance majeur pour une application communicante réside dans le modèle d'attente de l'exécution d'une requête de communication. Dans le cas de machines dotées de plusieurs processeurs ou simplement dotées d'interface de communication disposant d'une certaine autonomie, il est dans l'intérêt de l'application de pouvoir poursuivre ses calculs en parallèle à l'exécution des requêtes de communication. En particulier, le modèle d'attente de l'exécution d'une requête est déterminant du côté de la réception. Enfin dans le cas d'une approche client/serveur, où une requête émise déclenche une action du serveur, un choix est possible entre l'usage d'un processus servant ou l'emploi de mécanismes proches des *active messages*.

En théorie, lors de la conception d'une application, le choix du modèle de communication est une conséquence du type de problème que l'application cherche à résoudre. Une fois ce choix réalisé, les primitives de communication (bibliothèque de communication bas niveau ou environnement d'exécution communiquant) sont adaptées aux types de matériels disponibles ou souhaités.

Chapitre 4

Introduction aux systèmes à image unique

Dans le chapitre 1, nous avons brièvement introduit la notion de système à image unique. L'objectif visé par un système à image unique est de simplifier l'usage et la programmation d'une grappe de calculateurs. Cette simplification est réalisée par un ensemble de mécanismes de gestion globale des ressources présentes dans la grappe de calculateurs. L'enjeu des systèmes à image unique est d'effacer la notion de nœud et de réseau d'interconnexion derrière celle, bien connue des utilisateurs et des concepteurs d'applications, de machine parallèle.

La suite de ce chapitre est consacrée à une présentation détaillée des systèmes à image unique. Dans un premier temps, nous présentons différents modèles de systèmes à image unique. Puis nous étudions les aspects techniques de Kerrighed, un système à image unique conçu dans le cadre d'une activité de recherche du projet PARIS (IRISA/INRIA Rennes). Cette étude permettra de mieux déterminer les profils de communication dans un système à image unique.

4.1 Classification des systèmes à image unique

De nombreux projets de recherche existent dans le domaine des systèmes à image unique. Dans la suite de ce paragraphe, nous en présentons certains, les plus significatifs. Ces projets se distinguent par la méthode de gestion des ressources de la grappe, par le modèle d'application visé, et par le degré de transparence du SSI vis-à-vis des applications et des utilisateurs.

Les premières ressources à avoir été étudiées dans le cadre des systèmes à image unique sont : la mémoire, les processeurs ainsi que le sous-système disques. De nombreux projets ont rapidement offert une gestion globale de certaine de ces ressources.

Dans le prolongement de la gestion de processus dans les réseaux de stations de travail, de nombreux projets ont étudié la gestion globale de la ressource processeur dans un système à image unique. L'un des premiers systèmes fut Sprite[65], qui combine la gestion globale des processus à un système de fichiers distribués. Mosix[14, 5] et

Condor[53, 15] ont développé une architecture pour déplacer les processus d'un nœud à un autre de manière transparente pour le programme et l'utilisateur. Dans Mosix, la politique de placement/déplacement des processus est liée à la charge CPU et mémoire des nœuds. Le mécanisme d'équilibrage de charge travaille de manière à déplacer un processus d'un nœud chargé vers un nœud moins actif. Pour Condor, les politiques d'ordonnancement des processus dans la grappe lient l'information sur l'usage des CPUs à celle de l'usage du réseau afin de décider d'un déplacement de processus. Pour le système HiveOS[25], c'est la gestion globale de la mémoire qui est placée au centre du projet de recherche. Le système de mémoire est exploité comme base pour offrir une propriété de haute disponibilitée aux grappe de calculateurs.

Dans les systèmes précédemment évoqués, comme dans OpenSSI[40], l'approche défendue est de rapprocher la ressource logique du nœud où elle est exploitée efficacement : un processus est déplacé (ainsi que la totalité de son contexte mémoire) vers le processeur choisi par l'ordonnanceur de la grappe. Toutefois, en dehors des ressources mises en avant par ces systèmes (processeur ou mémoire) l'accès aux ressources logiques se fait de manière indirecte, en rejouant sur le nœud de création du processus, l'action effectuée à distance. Cet intermédiaire (appelé *home node* pour Mosix, *shadow process* pour Condor – figure 4.1), permet de masquer les déplacements du processus vis-à-vis de certaines ressources. Cette approche, bien qu'apportant toute la transparence nécessaire au bon fonctionnement d'un système à image unique, ne permet pas un fonctionnement efficace de celui-ci. De plus, un lien est créé, le temps de l'exécution du processus, entre le nœud recevant un processus et le nœud de départ de ce même processus. La défaillance d'un nœud pouvant donc entraîner la défaillance d'un autre nœud.

Le second axe de distinction des systèmes à image unique est le niveau de transparence souhaité du système à image unique par rapport à la machine SMP virtuellement créée. Des mécanismes tels que la gestion globale de la mémoire permettent d'accéder à une ressource depuis un nœud quelconque de la grappe. La question réside dans le placement de ces mécanismes au sein d'un système à image unique. Mosix a fait le choix de placer l'ensemble de ces mécanismes au sein d'une extension d'un système d'exploitation Linux. Cette approche permet une complète transparence pour l'ensemble des applications conçues pour le système d'exploitation Linux. En particulier, le principe du *home node* permet le déplacement de processus communiquant avec d'autre processus situés à l'intérieur ou à l'extérieur de la grappe de calculateurs. Du fait que l'ensemble des extensions sont réalisées dans le système d'exploitation, sans modification des interfaces système traditionnelles, la compatibilité avec les applications existantes est à la fois de niveau source et binaire. Le projet cJVM[7] relève d'une démarche similaire en modifiant non pas un système d'exploitation, mais en étendant une machine virtuelle JAVA[36]. Bien que l'exécution d'une machine virtuelle soit le plus souvent en espace utilisateur, cette machine virtuelle fait figure de machine et de système d'exploitation complet pour les applications JAVA. Hormis la contrainte liéée au langage JAVA, la transparence de cJVM vis-à-vis des applications est complète. Une dernière approche au niveau système d'exploitation est de proposer un nouveau modèle de pro-

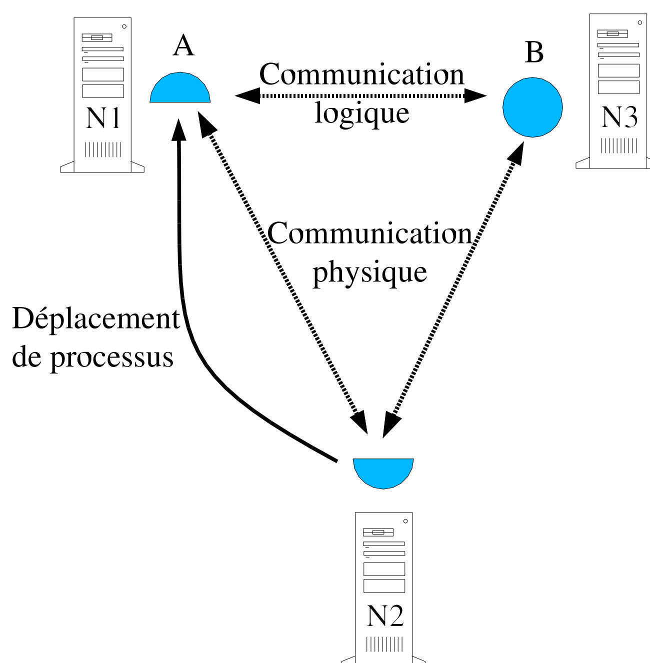


FIG. 4.1 – Principe du *home node*

grammation adapté à l'exécution sur grappe de calculateurs. C'est cette approche que le projet Plurix[35, 100] a retenu. Plurix est un système d'exploitation complet, conçu dans l'optique d'une exécution sur grappe de calculateurs. Plurix rend l'usage de la grappe transparent pour l'utilisateur comme pour le concepteur d'application, mais impose toutefois son propre modèle de programmation, fondé sur le langage JAVA, afin de décrire les tâches à exécuter ainsi que les contraintes entre ces tâches. Le système d'exploitation de la grappe a la responsabilité de déployer ces tâches en fonction de la disponibilité des ressources.

Toutes les approches précédentes se placent au niveau du système d'exploitation ou à un niveau analogue. Des travaux de recherche ont été menés pour développer des systèmes à image unique en tant qu'intergiciel déployé sur la grappe de calculateurs. Millipede[32] et Glunix[33] sont des intergiciels de ce type. L'un comme l'autre offrent une vision globale de la grappe de calculateurs mais doivent sacrifier le support de certains appels systèmes ou doivent sacrifier certaines formes d'optimisation. En effet, l'état des ressources manipulées étant interne au système d'exploitation, l'intergiciel ne peut y accéder directement et efficacement. Dans les deux projets évoqués, l'approche uniquement intergicielle a été enrichie d'un support spécifique du système d'exploitation de manière à fournir aux intergiciels les interfaces supplémentaires nécessaires.

4.2 Présentation d'un système à image unique : Kerrighed

Kerrighed est le système à image unique conçu et réalisé dans le cadre de l'activité de recherche Kerrighed dirigée par Christine Morin au sein du projet de recherche PARIS de l'IRISA/INRIA Rennes. Ce projet vise à offrir une machine virtuelle de type *SMP* au dessus d'une grappe de calculateurs homogènes. Il a débuté par le travail de thèse de Renaud Lottiaux sur la gestion globale de la mémoire[55, 58]. La gestion globale des processus au sein de la grappe a été étudiée par Geoffroy Vallée dans le cadre de sa thèse[96, 95]. Renaud Lottiaux et David Margery travaillent actuellement à de nouveaux développements au sein de Kerrighed, notamment sur la gestion globale des fichiers, et le support à OpenMP[56].

Kerrighed est développé en tant qu'extension du noyau du système d'exploitation Linux. En terme de développement, l'objectif est de concevoir l'infrastructure nécessaire au sein d'un noyau pour le déploiement des services de gestion globale de ressources. Par une telle approche, Kerrighed reste relativement indépendant des évolutions du noyau Linux, seule cette infrastructure étant à adapter. Nous disposons ainsi d'une plate-forme d'étude complète de système à image unique.

Architecture globale de Kerrighed Afin d'offrir la vision d'un système à image unique, Kerrighed gère les ressources système de la grappe au travers de plusieurs services distribuées inter-dépendants. Les ressources de la grappe sont à la fois physique telles que la mémoire, les disques durs et logique telles que les processus, les interfaces de communication. Par ailleurs, les services distribués permettent la mise en œuvre de services internes à Kerrighed tels la gestion d'identifiants uniques au sein de la grappe ou encore la gestion de verrous distribués. La figure 4.2 présente l'ensemble des services développés au sein de Kerrighed.

Deux groupes de service sont à distinguer. Chronologiquement, le SE met en place les services lui permettant d'être opérationnel et de communiquer avec les autres nœuds de la grappe de calculateurs. Dans un second temps, l'infrastructure pour les services distribués est mise en place, et les services distribués sont déployés.

Dans la suite de ce chapitre, nous détaillons les services distribués de gestion globale de la mémoire des processus. Ces services ont été choisis pour leur caractère crucial dans Kerrighed ainsi que pour leurs besoins complémentaires en matière de communication.

4.3 Gestion globale de la mémoire

De part son aspect central dans l'exécution d'un processus, la mémoire nécessite une attention toute particulière lorsqu'il est souhaité l'exploiter de manière globale et cohérente dans un système à image unique.

La suite de cette partie est consacrée à la gestion globale de la mémoire au sein de Kerrighed. Cette présentation est réalisée sous l'angle des besoins en communication.

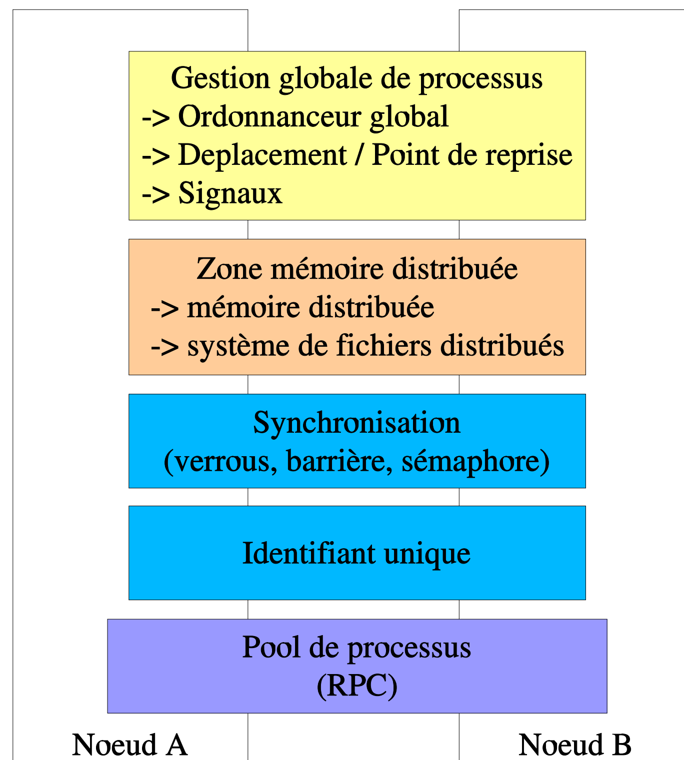


FIG. 4.2 – Services distribués de Kerrighed

4.3.1 Architecture de la mémoire

Le service distribué de mémoire vise à offrir une gestion globale des pages mémoire au sein de Kerrighed. Cette gestion est assurée par un mécanisme appelé **conteneur**[55]. Les conteneurs offrent la notion de zone mémoire à l'échelle de la grappe de calculateurs. À l'aide des conteneurs, une mémoire partagée répartie a été développée dans Kerrighed. Un conteneur est une unité mémoire abstraite globale et unique dans la grappe. Les conteneurs font partie intégrante du SE et s'intercallent entre les gestionnaires physiques et les gestionnaires virtuels traditionnels. La page mémoire constitue l'élément de base d'un conteneur. Un conteneur est relié à son environnement (logiciel comme matériel) par une entité appelée **lieur**. Les lieurs entre les conteneurs et les applications sont appelés des **lieurs d'interfaces**. Les lieurs entre les conteneurs et le matériel sont appelés des **lieurs d'entrée/sortie**.

Lors d'un défaut de page mémoire par une application, le mécanisme des conteneurs localise la page et transmet la requête au nœud disposant de la page. Cette dernière est transmise au demandeur après mise à jour des informations de gestion correspondantes (*i.e :* état de la page, liste des copies, etc.). La mise en œuvre est réalisée par un ensemble de processus présents sur chaque nœud de la grappe. Ces processus ont pour rôle de transmettre et de traiter les requêtes au sein du système de mémoire. Au final, les conteneurs assurent le partage des données entre les nœuds de la grappe.

Par conception même, les algorithmes assurant la gestion globale de la mémoire sont complexes à mettre en œuvre. Comme en atteste la figure 4.3 qui représente l'automate d'état associé à la gestion globale de la mémoire. Cet automate indique clairement que de nombreuses transitions sont provoquées par des messages extérieurs au nœud local. Ces messages correspondent à des requêtes échangées entre les nœuds participant à la gestion globale de la mémoire. Par conséquent, la latence correspondant au cheminement et à la remise des messages est un facteur important dans les performances globales du système de gestion de la mémoire, et par conséquent de Kerrighed dans son ensemble.

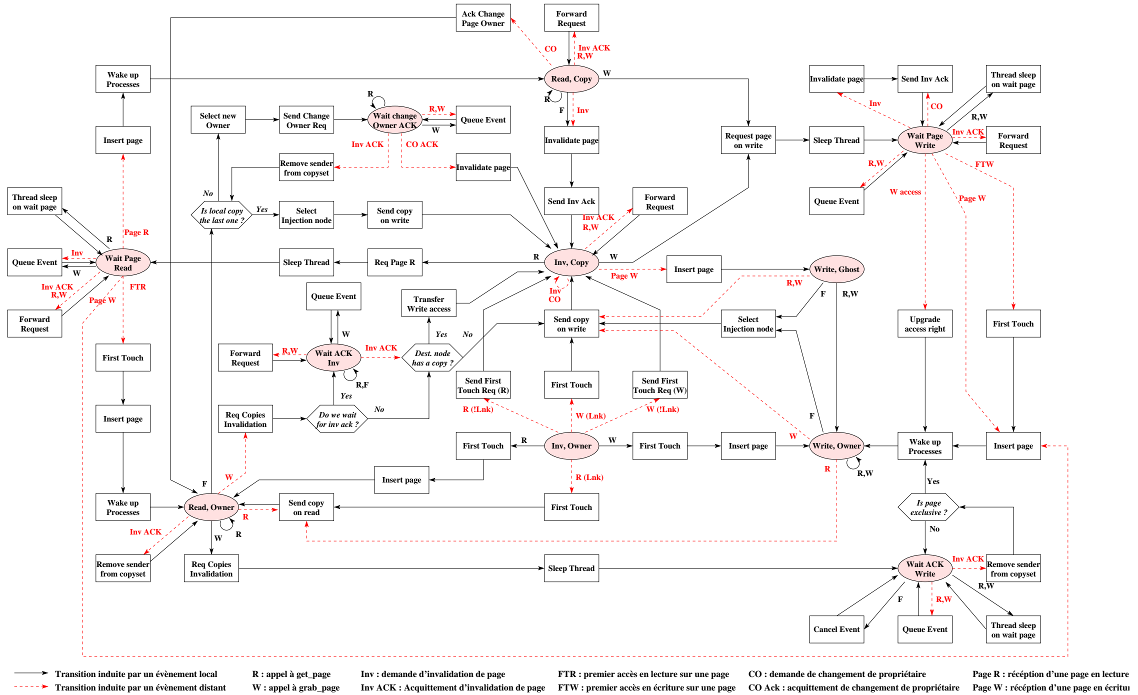


Fig. 4.3 – Automate de transition d'états des pages d'un conteneurs

De plus, afin de limiter cette complexité et éviter de nouveaux états dans un automate déjà bien chargé, de nombreuses hypothèses sont faites sur le système de communication sous-jacent. Les canaux de communication entre deux nœuds sont supposés fiables et ordonnés : les messages, pour un canal donné, ne peuvent pas se perdre et sont réceptionnés et délivrés dans le même ordre qu'ils sont émis.

L'aspect ordonné des canaux de communication ne simplifie pas uniquement la mise en œuvre des algorithmes. En effet, cette simplification va dans le sens d'une plus grande efficacité, car l'hypothèse de canal ordonné évite l'usage de mécanisme d'acquittement bien souvent coûteux en nombre de messages.

4.3.2 Profil de communication

Le graphique 4.4 représente le profil de communication d'une application parallèle simple, fondée sur le modèle de communication par mémoire partagée (Orthonormalisation de matrice selon l'algorithme de Grahm-Schmidt), exécutée sur Kerrighed.

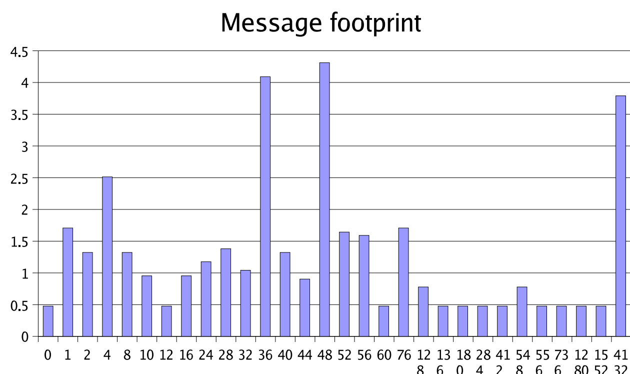


Fig. 4.4 – Empreinte de communication pour une application en mémoire partagée

Nous observons deux caractéristiques importantes. En terme de nombre de messages, 84% des messages sont de petite taille (<100 octets). Ces messages correspondent essentiellement aux requêtes pour la mise en œuvre des services distribués. En terme de nombre d'octets transférés, 95% des données l'on été au sein de messages légèrement supérieurs à 4 K-octets. Ces messages correspondent aux échanges de pages mémoire dans Kerrighed.

Lors des transferts de pages mémoire, le système de gestion globale de la mémoire fait l'hypothèse qu'une page émise sur le réseau peut être modifiée dès le retour de l'appel de fonction correspondant. Par conséquent, une copie de la page doit être réalisée afin de permettre l'émission (et éventuellement la ré-émission) de cette page. Cette approche a été nommée *send&forget*. Ultimement (et en acceptant une modification du système de gestion de la mémoire), cette approche peut être remplacée par une interface de communication associant une action en fin d'émission (effective) des données.

4.4 Gestion globale des processus

Lors de la prise en main d'un système à image unique, l'une des fonctionnalités les plus visibles réside dans la gestion globale des processus. En effet, pour des raisons de politique de gestion de la grappe de calculateurs, l'administrateur ou le système lui même, peuvent vouloir modifier la localisation des processus en cours d'exécution. Un ensemble de mécanismes destinés au contrôle de la grappe, aux prise de décision d'ordonnancement et aux déplacements des processus sont nécessaires.

Nous étudions dans la suite de cette partie l'architecture du système de gestion globale des processus ainsi que son impact sur le système de communication.

4.4.1 Architecture de la gestion des processus

Dans Kerrighed, l'architecture pour la gestion globale des processus repose sur deux axes distincts. Tout d'abord la partie technique consistant à extraire l'état d'un processus de manière à pouvoir le déplacer, le dupliquer ou encore le sauvegarder. De manière à avoir une gestion efficace des processus manipulés, le second axe permet la définition ainsi que l'exécution des politiques choisies (à l'aide d'un **ordonnanceur global**). Cette architecture est présentée par la figure 4.5.

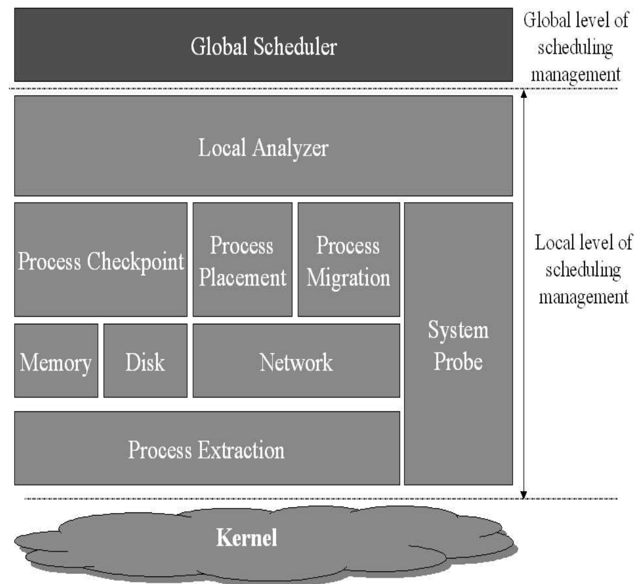


FIG. 4.5 – Gestion globale des processus

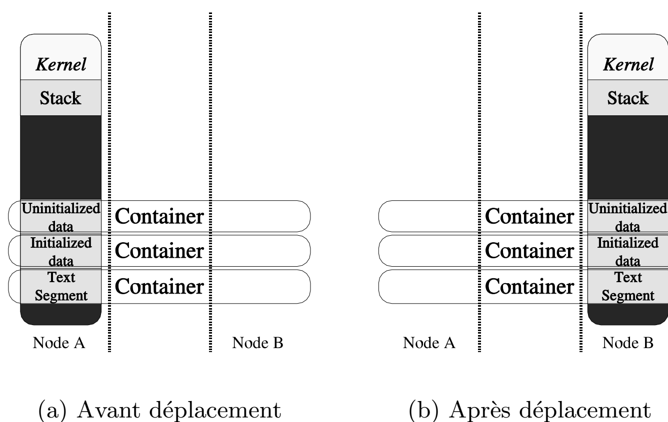
Ordonnanceur global à la grappe L'application des politiques de placement et d'exécution des applications se fait par l'ordonnanceur global. Les décisions sont prises sur la base d'informations relevées par un ensemble de sondes (charge processeur, température de boîtier, *etc.*). Ces informations sont une première fois traitées (et éventuellement combinées afin d'en créer de nouvelles) par un analyseur local à chaque nœud. Cet analyseur local transmet, éventuellement, l'information à l'ordonnanceur global. L'ordonnanceur global décide des placements ou des déplacement de processus à partir des informations transmises par l'ensemble des analyseurs locaux.

Déplacement de processus L'outil principal pour l'application des politiques d'ordonnancement est le déplacement de processus. Afin de déplacer un processus l'état de ce processus doit être capturé afin de pouvoir le recréer sur le nœud de destination. Cette capture d'état est sensiblement la même que pour la création d'un point de reprise. L'état d'un processus se compose principalement :

- des valeurs des registres du processeurs d'exécution,

– du descripteur de tâche (comprenant des informations sur les parentés du processus, l'état des signaux, *etc.*),
– des zones mémoire présentes dans l'espace d'adressage du processus,
– des descripteurs de fichiers.

Comme l'indique la figure 4.6, le mécanisme des conteneurs (paragraphe 4.3) est utilisé afin d'alléger le déplacement de l'espace d'adressage mémoire. Chaque zone mémoire est liée à un conteneur. Après déplacement du processus, une simple re-connexion aux différents conteneurs suffit à restaurer le contexte mémoire initial.



(a) Avant déplacement (b) Après déplacement

Fig. 4.6 – Conteneur liant des segments mémoire

Lors de la migration, seul le contenu des registres du processeur, le descripteur de tâche ainsi que les identifiants des conteneurs doivent être collectés et transmis au nœud de destination.

Cette approche, bien qu'efficace pour les applications écrites sur le modèle de partage de mémoire, ne permet pas le déplacement d'application faisant usage d'interfaces de communication par message telles que les *sockets*.

4.4.2 Profil de communication

Le déplacement d'un processus amène à la collecte et à l'envoi de nombreuses données de petite taille. Deux approches sont envisageables, soit pour chaque petite donnée, un message est transmis, soit les données sont recopiées afin de n'avoir qu'un seul (gros) message à transmettre. Le premier cas, entraîne une perte d'efficacité dans l'usage du réseau alors que le second cas impose de connaître à l'avance la taille du message final. Cette taille n'est disponible qu'après un parcours des structures de données représentant le processus, ce qui revient à faire une fausse migration dans le système d'exploitation.

4.5 Gestion globale des synchronisations

Le dernier exemple de service distribué que nous avons choisi de présenter est celui mettant en œuvre un mécanisme de verrou global au sein de Kerrighed. Les verrous mis en œuvre dans Kerrighed utilisent un algorithme à base de jeton. Pour chaque verrou, un jeton unique circule entre les nœuds de la grappe en fonction des demandes et des libérations de verrous.

Tout comme le service de gestion globale de la mémoire, ce service fait l'hypothèse de canaux de communication fiables et ordonnés. Les raisons de ce choix sont similaires à celles évoquées précédemment : simplification des algorithmes mis en œuvre. Toutefois, nous pouvons remarquer que si le système de communication ne gère pas cette propriété, de nombreux services devront disposer de mécanismes *ad hoc* pour pallier cette déficience.

Dans ce contexte particulier, la migration d'un processus détenant un verrou entraîne la migration du verrou. Des problèmes de concurrence entre les demandes de verrous et la migration peuvent survenir le temps de la restauration du verrou sur le nœud de destination. Plus que l'ordre de remise des messages, l'ordre d'exécution des requêtes doit être strictement séquentiel (et sans recouvrement) entre certaines requêtes. Dans le cas contraire, une requête de demande de verrou pourrait être traitée sur un nœud n'ayant pas achevé la restauration du verrou considéré.

Enfin, le second outil de synchronisation offert par ce service est celui de la barrière. Le principe de la barrière est simple et consiste à s'assurer que tous les processus participant à une barrière se trouvent dans la même situation (en les bloquant) avant de libérer ces processus. La libération des processus est réalisée par une diffusion fiable d'une requête de libération à l'ensemble des nœuds hébergeant au moins un processus participant à la barrière. Dans le cadre d'applications conçues sur le modèle de mémoire partagée, un environnement de programmation tel que OpenMP fait un usage intensif des barrières (généralement à chaque itération de la boucle de calcul). Par conséquent, les performances de la primitive de communication par diffusion affectent fortement celles de l'application à mémoire partagée usant de barrières.

4.6 Synthèse

L'étude de Kerrighed a permis d'établir certaines des caractéristiques qui semblent nécessaires lors de la conception d'un système de communication pour un système à image unique.

Les interfaces de communication ne peuvent être restreintes au simple envoi/réception de messages. Le service de gestion globale de la mémoire fait l'hypothèse que les pages mémoire envoyées sur le réseau peuvent être modifiées des le retour de fonction. Cela entraîne l'usage d'interface de type *send&forget*. La contrainte principale de cette interface, est qu'elle force la copie immédiate des données qui lui sont confiées. Toutefois une évolution de ce service, pourrait utiliser avantageusement un mécanisme permettant de différer cette recopie.

D'autre part, nous notons la présence de nombreux petits messages (comme dans une migration) ou de messages construits avant d'être envoyés forçant à des recopies (en particulier lors de l'envoi de pages mémoire auquel est associé un en-tête décrivant la page). De tels schémas de communication pourraient se satisfaires de mécanisme fondés sur l'interface *pack/unpack* de Madeleine[11].

Enfin, la structure même des services distribués développée sur la base de RPC amène à envisager l'optimisation de ce type de communication. En effet, à l'émission, différents en-têtes sont ajoutées de manière à constituer le message : une interface de type *pack* résoudrait le problème. À la réception, le message déclenchant une requête doit être exécuté le plus rapidement possible : les *active message*[98] constituent une solution pour réduire les latences.

Par ailleurs, les processus communiquant par message nécessitent un support spécifique analogue à celui offert par les conteneurs pour le déplacement efficace de zones mémoire. Un tel support doit tenir compte de la volonté d'établir des points de reprise d'applications conçues sur le modèle de communication par échange de messages.

De part sa conception, un système à image unique dépend entièrement de son réseau d'interconnexion. Par conséquent la haute disponibilité d'un tel système se doit de reposer sur un support dédié du système de communication. Ce système de communication doit pouvoir offrir une architecture permettant de remonter des informations de changement de configuration aux services système concernés. Ces changements de configuration concernent :

- l'ajout et le retrait de liens de communication au sein de la grappe,
- l'ajout et le retrait de nœuds,
- la détection de la défaillance d'un lien ou d'un nœud.

Chapitre 5

Système de communication pour un système à image unique

Le concept de système à image unique est une approche particulièrement élégante pour aborder le problème de la gestion efficace des grappes de calculateurs. Cette approche se propose d'offrir la vision d'une machine unique aux utilisateurs et concepteurs d'applications parallèles, à partir de l'ensemble des nœuds de la grappe de calculateurs. En particulier, nous nous intéressons aux SSIs fournissant l'illusion d'une machine multi-processeurs à mémoire partagée. Ainsi, les applications déployées sur un système à image unique peuvent être conçues selon le paradigme de communication par mémoire partagée ou selon celui de communication par échange de messages. Un système à image unique tel que Kerrighed est construit autour d'un ensemble de services système distribués s'exécutant sur les nœuds de la grappe de calculateurs.

L'architecture distribuée d'une grappe de calculateurs donne un rôle de premier plan au système de communication et en particulier au réseau d'interconnexion des nœuds. Ce rôle est double. D'une part, le système de communication doit permettre une exécution efficace des applications communiquant par échange de messages, y compris après déplacement d'un ou plusieurs processus de cette application entre les nœuds de la grappe. D'autre part, ce système de communication doit offrir un modèle de communication adapté à la réalisation de services système distribués.

Dans la suite de ce chapitre, nous réalisons une synthèse des caractéristiques des systèmes de communication dans les grappes de calculateurs. Cette synthèse supporte les idées directrices d'un système de communication apportant une solution complète et homogène aux problèmes identifiés. Le concept de **transaction de communication** décrit un modèle de communication dédié à la conception et la réalisation de services système distribués. Puis nous introduisons le concept de **flux dynamiques**, qui fournit le support nécessaire aux applications communiquant par échange de messages. Enfin, nous présentons l'architecture globale du système de communication résultant de nos travaux.

5.1 Fondement pour un service de communication dédié aux systèmes à image unique pour grappe de calculateurs

Les grappes de calculateurs sont constituées de machines interconnectées par un réseau. Afin de simplifier l'utilisation et l'administration de ce type de machines distribuées, le concept de système à image unique a été proposé et développé [14, 33, 7, 53, 47]. L'objectif des systèmes à image unique est d'offrir une machine virtuelle multiprocesseurs à mémoire partagée, à partir d'une grappe de calculateurs homogènes. Du fait de l'architecture d'une grappe, le système de communication a un rôle essentiel dans les performances globales de ces machines. Toutefois, à notre connaissance, aucun système de communication optimisé pour les communications de noyau à noyau n'est disponible. En effet, jusqu'à l'arrivée de systèmes d'exploitation distribués, seuls des files d'exécution en espace utilisateur communiquaient avec d'autres files d'exécution sur d'autres machines. Par conséquent, la conception d'un système de communication dédié aux échanges de messages de SE à SE ne revêtait que peu d'intérêt. Par ailleurs, les SSI tels que Mosix et OpenSSI ont préféré axer leurs travaux de recherche sur la gestion globale de processus et la vision uniforme de la grappe. En conséquence, ces SSI ont utilisé le système de communication TCP/IP tradionnel de Linux afin de concevoir un mécanisme simplifié de type RPC.

Notre objectif est de concevoir un système de communication adapté aux besoin d'un système à image unique. En premier lieu, le système de communication doit permettre la création, l'acheminement et le traitement de messages au sein des services système distribués de manière efficace. Sans cette première propriété, l'ensemble des services système, et par conséquent les applications, seraient pénalisés. La seconde propriété est celle du multiplexage de la ressource correspondant au système de communication, afin d'en permettre des accès concurrents par les services système distribués. Enfin, le support aux applications doit être le plus transparent possible afin d'éviter toute modification du code source des applications, pour les exécuter sur un SSI. En particulier, un support distribué doit être fourni aux interfaces de communication standard que sont les *socket (INET ou UNIX)*, les *pipe*, etc.

L'aspect performance est particulièrement important pour deux raisons. Tout d'abord, la notion de service distribué repose sur le système de communication. Si celui-ci est inefficace, c'est l'ensemble du SE distribué qui est pénalisé. Par ailleurs, un système de communication adapté aux applications doit être offert afin que les mécanismes de gestion globale des processus puissent manipuler (placer, déplacer et effectuer des points de reprise) les processus d'applications parallèles conçues selon le modèle de *communication par échange de messages*.

Afin d'étudier précisément les caractéristiques d'un système à image unique, nos travaux ont été conduits dans le cadre de Kerrighed, mais nous pensons qu'ils peuvent être adaptés à d'autres systèmes à image unique.

Les modèles d'application visés pour un système à image unique sont la programmation par mémoire partagée ainsi que la programmation par échange de messages. Le

modèle de programmation par mémoire partagée repose sur l'usage d'un mécanisme logiciel ou matériel réalisant effectivement un partage de la ressource mémoire entre plusieurs nœuds. Dans un tel modèle, l'usage du réseau d'interconnexion des nœuds de la grappe de calculateurs est masqué par le service de partage de la mémoire. Les échanges de données sur le réseau sont implicites pour l'application. Dans le modèle de programmation par échange de messages, les échanges de données sont explicites pour l'application.

Au sein d'un système à image unique, la gestion globale des ressources a pour but d'optimiser l'utilisation globale de ces ressources. Un mécanisme de déplacement des processus permet la réalisation de politiques de placement de processus ainsi que d'équilibrage de charge. En particulier, un support au déplacement de processus communiquant par message est nécessaire. De même un support spécifique est nécessaire pour réaliser des points de reprise d'applications communiquant par message. Enfin, le système de communication représentant le cœur d'un système à image unique, il est nécessaire qu'il dispose de primitives de haute disponibilité afin de gérer, dans les modèles de communication, l'ajout, le retrait et la défaillance d'un nœud dans la grappe de calculateurs.

5.2 Modèle de communication d'un service système distribué

Un système à image unique tel que Kerrighed repose sur un ensemble de services système distribués. Ces services sont présents sur l'ensemble des nœuds composant la grappe de calculateurs. Chaque instance locale a un nœud d'un tel service lui permettant d'agir sur une ressource pour le compte du nœud lui-même, ou d'une instance du même service sur un autre nœud. Les communications entre instances d'un même service distribué sont faites par un mécanisme d'appel de procédure à distance.

Par conception, la majeure partie des messages échangés dans un service système distribué peuvent se classer en deux catégories : les messages de requêtes ainsi que les réponses à ces requêtes. Les performances des services distribués dépendent de la rapidité du traitement de ces requêtes. Par conséquent, afin d'assurer le fonctionnement d'un système à image unique, ces requêtes doivent pouvoir être traitées prioritairement par rapport à l'ensemble des autres processus de la grappe. En particulier, le délai entre la réception physique d'une requête et son traitement effectif doit être minimal. Les messages de requête correspondent donc au modèle de communication par message actif présenté dans le paragraphe 3.2.

Par ailleurs, le message d'une requête peut être composé de plusieurs éléments. Par exemple, lors du déplacement d'un processus, de nombreuses structures de données doivent être assemblées afin de créer l'image du processus à déplacer. En considérant uniquement cet aspect, nous reconnaissons un modèle de composition de messages tel que présenté dans le paragraphe 3.2.

Idéalement, reprenons l'exemple de la migration d'un processus. Nous souhaitons que le nœud d'origine du processus puisse construire l'image du processus, sans utiliser

de tampon intermédiaire qui induirait une recopie pouvant nuire aux performances du déplacement de processus. Par ailleurs, il est souhaitable que le récepteur commence la reconstruction du processus par étape en parallèle avec l'émission. Par conséquent, nous souhaitons combiner les avantages de la composition de message à ceux du modèle de message actif. L'objectif est de permettre un recouvrement maximum entre le processus émetteur, le processus récepteur et le système de communication lui-même.

Enfin, les technologies d'interconnexion des grappes de calculateurs étant variables, il n'est pas raisonnable de concevoir un modèle de programmation pour un type de matériel en particulier. Par conséquent, un système de communication pour système à image unique se doit d'être le plus indépendant possible du matériel afin de répondre à des critères de portabilité et de ne pas dépendre d'un matériel ou d'un protocole en particulier.

5.3 Modèle de communication pour une application communiquant par échange de messages

Synthétiquement, une communication entre deux processus consiste à transmettre un ensemble de données vers l'adresse du processus récepteur. Dans le contexte des systèmes à image unique, il est donc possible de distinguer plusieurs types de communication selon la localisation des processus communicants.

Définition 23 ***Communication extra-grappe****: Les* ***communications extra-grappe*** *sont toutes les communications entre une grappe et une machine quelconque en dehors de la grappe.*

Définition 24 ***Communication inter-grappe****: Les communications inter-grappe sont toutes les communications entre deux grappes distinctes exécutant chacune une instance du même système d'exploitation.*

Définition 25 ***Communication intra-grappe****: Les communications intra-grappe sont toutes les communications à l'intérieur d'une grappe de calculateurs dotée d'un SSI.*

La principale différence entre ces trois types de communication réside dans la nature des hypothèses que nous pouvons réaliser. D'un coté, les communications intra-grappe sont réalisées dans un environnement logiciel que nous maîtrisons parfaitement : seule une compatibilité de niveau interface de communication est nécessaire. Au contraire, dans le cas des communications extra-grappe, nous ne maîtrisons qu'une seule des deux parties en communication : le respect des protocoles de communication est nécessaire. Dans la suite de ce chapitre nous nous intéressons uniquement aux communications intra-grappe. Le traitement des communications inter-grappe et extra-grappe est détaillé dans le chapitre 8.

Notre problématique est de fournir un support efficace au mécanisme de déplacement de processus ainsi qu'à la réalisation de points de reprise des processus communicants. Comme nous le montre le chapitre 2, il n'est pas possible d'identifier un composant

logiciel ou matériel commun à l'ensemble des processus communiquant par échange de messages. Aucun language, aucune bibliothèque de communication, aucun protocole de communication, aucune technologie particulière ne fait autorité pour les processus communicants. Ce constat amène à plusieurs conclusions. Tout d'abord, le système de communication d'un système à image unique ne doit pas dépendre d'un type de matériel de communication particulier. De manière analogue, le support des interfaces de communication doit être le plus vaste possible. Les interfaces classiques dans le monde des grappes de calculateurs doivent être disponibles (notamment l'interface socket inet). Afin de fournir la vision d'une machine SMP sur la grappe de calculateurs, les interfaces classiques aux machines SMP (socket unix, pipe, etc.) doivent être étendues dans une version distribuée sur la grappe. Enfin, des applications parallèles employant des interfaces de communication spécialisées telles que la bibliothèque GM devraient pouvoir bénéficier des propriétés de l'architecture du système de communication mis en place pour le déplacement et la réalisation de points de reprise.

Toutefois, il ne faut pas perdre de vue, que pour bon nombre d'applications, l'objectif premier est la performance. Par conséquent, une indirection dans les communications afin de permettre le déplacement d'un processus en cours d'exécution n'est pas acceptable car trop coûteuse.

Traditionnellement, une communication par échange de messages débute par la localisation mutuelle des processus communicants. Une fois cette localisation établie, elle n'est plus remise en cause jusqu'à la fin de la communication. L'usage d'une indirection par le nœud d'origine d'un processus déplacé permet de ne pas avoir à modifier (explicitement) la localisation des interlocuteurs. Nous proposons d'introduire un mécanisme de localisation dynamique des processus communicants dans les protocoles de communication existants.

En se restreignant aux communications intra-grappe, il a été de nombreuses fois démontré que le remplacement complet de la pile réseau du système d'exploitation par une pile dédiée aux communications internes à la grappe homogène de calculateurs, améliore grandement les performances des applications communiquant par échange de messages [29, 43, 39]. En effet, les nœuds d'une grappe de calculateurs étant généralement sur le même réseau physique, il est possible d'alléger la pile de communication de divers mécanismes tels que ceux permettant la gestion d'architectures hétérogènes.

5.4 Transaction de communication

L'étude des services système distribués présents dans Kerrighed permet de dégager deux caractéristiques du profil des communications. Tout d'abord, ces services système distribués sont conçus selon un modèle d'appel de procédure à distance. En effet, le schéma classique d'exécution consiste pour un nœud à demander à un autre nœud la réalisation d'une action sur une ressource déterminée, cette opération pouvant éventuellement être une demande de transfert physique (d'un nœud à un autre) de la ressource logique (par exemple, lors du déplacement d'une page mémoire). La seconde

caractéristique concerne le fait que les messages sont fréquemment composés de plusieurs données distinctes présentes sur le nœud émetteur.

Le concept de **transaction de communication** que nous proposons a pour but de combiner la souplesse de création/réception des messages de l'architecture Madeleine[11] à l'efficacité des appels de procédure à distance proposée par les *Actives Messages*[98].

Une transaction de communication permet de décrire le contenu d'un message par une succession de requêtes de communication. Chacune de ces requêtes peut être associée à une action destinée à être exécutée au plus tôt lorsque que la requête de communication est réalisée. En particulier, ces actions se placent autant que possible, dans le contexte simplifié d'un signal matériel annonçant la réalisation de l'arrivée ou de l'envoi effectif d'un message sur le nœud correspondant. Ces actions peuvent à leur tour, créer ou modifier des transactions de communication. Par ailleurs, une action associée à une réception peut terminer prématurément le traitement de la transaction et ainsi annuler les réceptions à venir dans cette transaction. Cette caractéristique permet, par exemple, en cas de surcharge de la machine, de ne plus proceder au traitement de la transaction en cours. Le rôle des transactions de communication est d'offrir un contexte commun de communication, de manière à lier le traitement de plusieurs fragments faisant partie du même message.

Ainsi le modèle de communication proposé combine un modèle de composition de messages et un modèle de composition des actions déclenchées par les événements de communication.

5.5 Flux dynamiques

Lors de la conception d'une application parallèle suivant le modèle d'échange de messages, plusieurs flux de données sont définis. Un flux de données matérialise les communications existant entre deux processus de l'application parallèle. Une application emploie un flux au travers d'interfaces de communication mettant en œuvre un protocole particulier : *soket Inet*, *socket unix*, *pipe*, etc. Lors du déplacement d'un processus, la propriété de transparence du système à image unique fait que le flux doit être maintenu sans intervention du processus. Bien que répondant à ce critère de transparence, la méthode par indirection (par exemple les *home node* de Mosix, chapitre 4) pénalise l'ensemble des performances de toutes les communications intervenant après la migration du processus.

Au sein d'un flux de données, le problème à résoudre pour une extrémité, est de localiser efficacement l'autre extrémité du flux. Le concept de **flux dynamique** offre l'infrastructure permettant à une extrémité de localiser en permanence la ou les autres extrémités de son flux. Ainsi, à tout moment, les échanges de données se font directement entre le processus émetteur et le processus récepteur.

Par conception, le surcoût lié aux flux dynamiques est placé dans la phase de création du flux et dans la phase de déplacement d'une extrémité. En revanche, nous faisons le choix de conserver un mécanisme d'échange de données aussi performant que possible dans le respect des interfaces standard fournies aux applications.

5.6 Architecture globale du système de communication

La contribution de cette thèse est la conception et la réalisation d'un système de communication intégrant les concepts présentés dans les paragraphes précédents. Une tel système de communication apporte aux SSIs le support nécessaire aux déplacements des processus communicants (permettant ainsi l'utilisation de politique d'ordonnancement dynamique) et fournit un modèle simple de programmation de ses services distribués.

La figure 5.1 schématise l'architecture globale du système de communication proposé. Le système de communication que nous proposons dispose d'un niveau d'abstraction des technologies de communication (GLOIN). Cette abstraction rend le système de communication indépendant des technologies de réseau. À partir de cette abstraction de bas niveau, le modèle de communication destiné aux services systèmes distribués est mis en œuvre (GIMLI). Le service système des flux dynamiques (PALANTIR) est déployé en exploitant le modèle de communication des services systèmes distribués. Finalement, les interfaces de communication standard sont offertes à partir des flux dynamiques, de manière à ce que les applications communiquant par message puissent bénéficier des flux dynamiques sans modification du binaire de l'application.

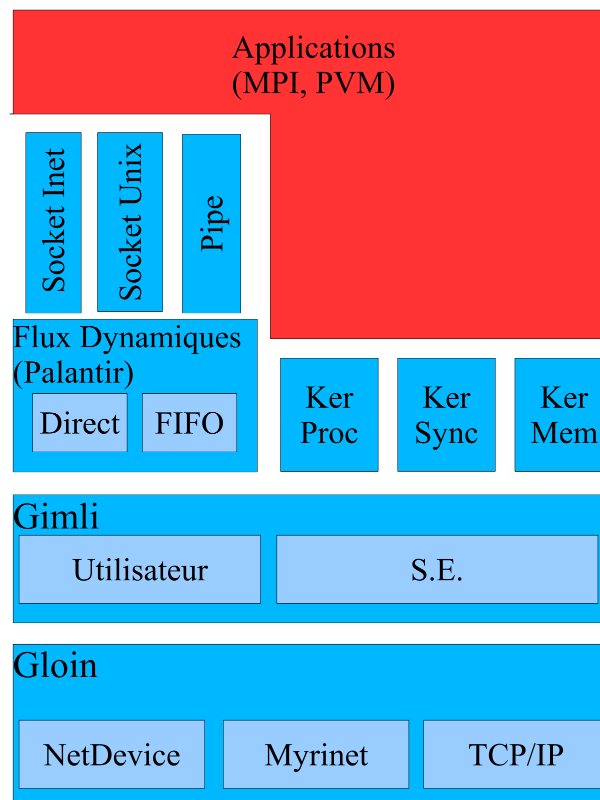


FIG. 5.1 – Un système de communication pour les applications d'un système à image unique

Chapitre 6

Architecture de communication pour un système à image unique

L'étude du système de communication pour grappe de calculateurs, nous amène à étudier deux problèmes : les communications au sein de services système distribués et les communications entre processus mobiles au sein de la grappe. Dans le premier cas, la latence entre l'émission d'un message et son traitement effectif est privilégiée. Dans le second cas, les performances de communication entre les processus mobiles doivent être indépendantes de tout déplacement de processus.

Dans un premier temps (paragraphe 6.1), nous dégageons les propriétés imposées par les technologies existantes ainsi que celles relevant des problèmes envisagés. À partir de ces propriétés nous présentons notre modèle général de communication au sein d'un SSI (paragraphe 6.2). Par la suite, nous détaillons un modèle de matériel (paragraphe 6.3) ainsi qu'un modèle de protocole (paragraphe 6.4) de communication sur lesquels sont fondés les transactions de communication (chapitre 7) et les flux dynamiques (chapitre 8). Finalement, le paragraphe 6.5 détaille l'ensemble des propriétés offertes pour la conception de protocoles de communication.

6.1 Caractéristiques des matériels et des protocoles de communication envisagés

Bien que théoriquement indépendantes, les solutions à ces problèmes font nécessairement appel aux mêmes mécanismes fondamentaux, tels que le dialogue avec l'interface de communication. L'objectif de ce chapitre est de déterminer quelle base commune peut être établie lors de la conception de protocoles de communication.

Profil des interfaces matérielles de communication Au delà des protocoles d'utilisation d'un matériel de communication, les contraintes de programmation sont déterminées par les spécifications matérielles de la machine. Ainsi, l'ajout de matériel à une machine de type PC, se fait principalement par l'enfichage d'une carte d'extension sur un bus de communication de type PCI[77, 76].

Deux types de mécanismes existent pour le transfert de données entre la mémoire centrale d'une machine et une carte d'extension : le mode PIO (*Programmable Input/Output*) et le mode MMIO (*Memory Mapped Input/Output*). Pour des raisons de performance et de souplesse d'utilisation, le mode MMIO est devenu le mode recommandé lors de la conception de matériel d'extension PCI. Grâce au MMIO, les échanges de données peuvent se résumer à de simples recopies de mémoire (centrale) à mémoire (embarquée). De plus, une telle opération de recopie peut être sous-traitée à une unité spécialisée : le contrôleur DMA (*Direct Access Memory*). Ces opérations de transfert entre mémoires, réalisées au niveau physique, imposent l'usage de plages de mémoire contigües. Par ailleurs, l'accès à des adresses mémoire réservées (appelées *registres*) des cartes d'extension permet le contrôle de la carte par le processeur principal (et par conséquent le système d'exploitation).

Outre la définition de registres, la norme PCI prévoit un mécanisme permettant à un matériel d'extension d'informer le processeur central de l'occurrence d'un événement : les *interruptions*. Comme son nom le suggère, ce mécanisme permet d'interrompre le processeur principal dans sa tâche courante afin d'exécuter un code de traitement de l'événement : par exemple procéder à la recopie d'une trame fraîchement reçue vers la mémoire centrale de la machine.

De manière plus spécifique aux matériels de communication, ces interfaces disposent d'identifiants matériel uniques. Ces identifiants de bas niveau permettent de distinguer les interfaces de communication, et sont employés lors de l'acheminement des trames entre les interfaces de communication. En général, sur un lien physique reliant deux interfaces de communication, les trames sont ordonnées mais la propriété de fiabilité (perte ou corruption de trame) n'est pas systématique. La communication est généralement de type point-à-point, cependant certaines technologies offrent un support matériel à la diffusion de trame. Enfin, certaines normes de communication, telles que l'Ethernet, imposent une taille de trame minimale et maximale.

Profil général d'un protocole de communication pour le SE Un service système (pour Linux) se caractérise par un accès privilégié à la ressource mémoire. En particulier, les zones de mémoire allouées disposent d'une adresse physique statique. De plus, la mémoire allouée peut être explicitement demandée comme contigüe.

Dans notre contexte de SSI, les communications de SE à SE sont des communications internes aux services distribués. Un service distribué étant déployé sur l'ensemble des nœuds présents dans la grappe de calculateurs, un mécanisme d'adressage des nœuds doit être mis en place. De plus, chacun des messages transmis représente un événement système. Par conséquent, il est facile d'imaginer que ces messages ne peuvent être ni perdus, ni même acheminés dans un ordre différent de celui de leur émission. Ces transferts de message sont de type point-à-point ou de type diffusion.

Enfin, lors de la création de messages, un en-tête de message, lié au protocole de communication, doit être ajouté. Lors de la réception, le traitement associé aux messages doit être réalisé le plus tôt possible afin de ne pas pénaliser l'ensemble de la grappe.

Profil général d'un protocole de communication pour des applications Par opposition aux services systèmes, une application quelconque ne peut généralement pas imposer que ses pages mémoire disposent d'adresses physiques statiques ou contigües. Il existe des techniques pour contourner ces contraintes, mais dans le cas général, celles-ci ne sont pas applicables.

Si nous considérons les différents protocoles de communication disponibles dans un SE de type Unix, il apparaît que la fiabilité du protocole n'est pas toujours nécessaire. Des protocoles tels que TCP, les pipe ou les FIFO assurent un service fiable, alors que le protocole UDP n'offre aucune garantie. Dans tous les cas, ces protocoles offrent directement ou indirectement un mécanisme d'adressage des nœuds de la grappe ainsi que des processus communicants.

Selon le protocole de communication, un processus en attente de réception peut se trouver bloqué ou en attente d'un signal. Lors de la réception du message, le protocole réalise l'opération permettant au processus de poursuivre son traitement (envoi de signal ou libération du processus bloqué).

Enfin dans le cadre d'applications parallèles communiquant par échange de messages, aucune hypothèse particulière ne peut être réalisée sur la taille des messages.

Synthèse À l'étude de ces profils, il apparait que certaines propriétés de communication sont commune aux deux profiles. Ces propriétés sont notamment :

- la fiabilité du canal de communication (pas de perte de message),
- l'intégrité du canal (pas de corruption de message),
- l'ordre d'acheminement des lessages,
- la diffusion de certain messages.

Une des caractéristiques de ces propriétés est qu'elles ne sont pas spécifiques à un protocole particulier.

6.2 Modèle de communication au sein d'un SSI

À partir des conclusions du chapitre 5, l'architecture du système de communication d'un SSI doit permettre la conception de protocoles à la fois pour les applications pour l'usage interne du noyau. Le paragraphe 6.1 montre que plusieurs protocoles partagent les mêmes besoin (fiabilité, intégrité, etc.). Il paraît donc souhaitable d'éllaborer des mécanismes communs, disponible dans une infrastructure de protocole, et indépendant de tout protocole particulier. Notre but est donc de réaliser une infrastructure offrant un ensemble de composants disponibles pour la conception de protocoles aussi divers que ceux correspondant au déplacement des extrémité d'un flux ou à l'efficacité des appels de procédure à distance.

La figure 6.1 représente l'ensemble d'une pile réseau orientée système à image unique. Cette figure illustre notre volonté de partir du paquet de données manipulé par les matériels de communication, pour arriver à la notion de flux classique reliant des processus communicants.

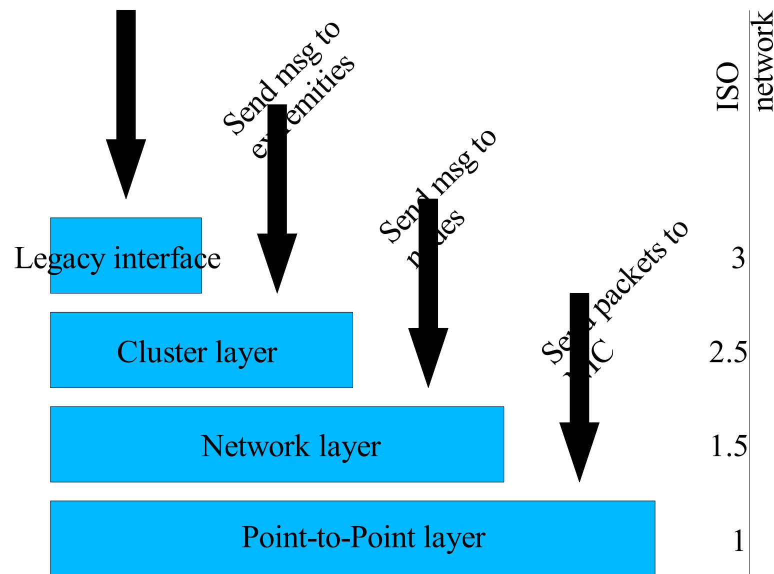


Fig. 6.1 – Système de communication de Kerrighed

Communication point à point Au niveau le plus bas, se trouve l'interface avec le matériel de communication. Ce niveau correspond à l'envoi et la réception de paquets de données d'une interface réseau vers une autre, au sein de la grappe de calculateurs. Le niveau de communication point à point permet d'offrir une abstraction des matériels de communication. Par ce niveau est reçu l'ensemble des messages en provenance des autres nœuds. Par conséquent, une surveillance passive de l'activité des nœuds (ainsi que des liens de communication) est possible simplement en notant les trames reçues. Ce niveau permet la détection des défaillances de liens (ou de sa propre interface de communication).

Communication entre nœuds En assemblant les paquets, le système de communication réseau permet la reconstitution et le traitement des messages. À ce niveau, les nœuds communiquent entre eux à l'aide de messages indépendants du matériel d'interconnexion employé. Lorsqu'à ce niveau, l'ensemble des liens de communication vers un nœud sont détectés comme défaillants, le nœud correspondant peut être considéré comme suspect et une procédure spécifique de défaillance (ou de retrait) peut débuter. L'assemblage du système de communication point à point et du système de communication entre nœuds représente notre infrastructure pour la conception de protocoles de communication. À ce niveau, les nœuds ainsi que les communications entre les nœuds sont supposés fiables. Dans notre contexte, une communication fiable signifie que le message transféré est bien reçu et bien traité. En cas de défaillance, une action extérieure au système de communication doit être déclenchée afin de corriger la situation.

Communication entre extrémités de flux Les flux de données circulent d'un processus à l'autre par l'intermédiaire d'interfaces de communication nommées **extrémités de flux**. Dans notre contexte, ce sont ces extrémités qui peuvent être déplacées d'un nœud à l'autre ou peuvent faire partie d'un point de reprise. Les communications ne sont donc plus de nœud à nœud mais d'extrémité mobile à extrémité mobile.

Communication entre interfaces standard À l'aide des extrémités de flux précédemment construites, il est possible d'offrir une version des interfaces standard de communication qui soit compatible avec les propriétés souhaitées d'un système à image unique.

Cette architecture permet de définir une pile réseau dédiée aux systèmes à image unique qui soit indépendante du type de matériel de communication (au gestionnaire de périphérique près) et des protocoles de communication des applications.

6.3 Modèle de matériel de communication

Le modèle de matériel de communication correspond à l'abstraction du niveau de communication point à point. Ce niveau a pour but :

- d'identifier matériellement, les interfaces de communication présentes sur le même réseau d'interconnexion,
- de mettre en forme les trames de communication avant leur émission,
- d'émettre une trame sur le réseau d'interconnexion,
- de permettre l'extraction du contenu d'une trame reçue.

Ce modèle, très simple, se focalise principalement sur l'échange de données entre deux interfaces de communication. La synthèse des propriétés de communication établie dans le paragraphe 6.1, nous permet de définir notre modèle de matériel de communication. Il s'agit de définir le matériel de communication de manière à pouvoir réaliser de manière matériel tout ou partie des propriétés de communication.

Dans notre architecture, le modèle de matériel de communication permet la réalisation de communication fiable (intégrité du contenu et absence de perte de trames) et ordonnée. De plus, un support à la diffusion (fiable) est requis pour certains services systèmes. Enfin, un mécanisme de contrôle de flux est nécessaire pour atteindre un bon niveau de performance en évitant de saturer les interfaces de communication.

Les matériels de communication n'offrent pas tous les mêmes caractéristiques lors des échanges de données. Lors de l'initialisation du périphérique et de son intégration dans notre architecture de commmunication, il est possible de définir les propriétés du modèle non prises en charge par le matériel :

- **GLOIN_DEV_NEED_CRC** : Absence de garantie de l'intégrité des trames de communication,
- **GLOIN_DEV_NEED_FLOW_CTL** : Absence de mécanisme de contrôle de flux,
- **GLOIN_DEV_NEED_ACK** : Absence de garantie contre la perte de trames,
- **GLOIN_DEV_NEED_SEQ** : Absence de garantie sur l'ordre des trames,

– **GLOIN_DEV_NEED_BROADCAST** :Absence de diffusion de trames.

De manière à être relativement indépendant des choix technologiques, nous avons déterminé les primitives nécessaires à la réalisation d'une interface réseau Kerrighed. L'objectif est d'établir une interface de programmation claire pour la réalisation de nouveaux gestionnaires de périphérique.

À titre d'exemple des propriétés d'une primitive de communication, considérons deux primitives de communication différente : une carte Ethernet et le protocole de communication TCP. Dans ce cas, aucune des propriétés retenues n'est garantie par l'interface l'infrastructure de communication : une alternative indépendante des protocoles est fournit par l'infrastructure de protocol.

Ces primitives concernent l'envoi et la réception de données.

send_msg La fonction send_msg est la fonction d'envoi de messages. Selon la *MTU* du périphérique de communication, le message est éventuellement fragmenté. Les informations propres au matériel de communication (par exemple l'en-tête Ethernet) sont ajoutées pour créer les paquets de données. Puis, selon les propriétés du paquet, il est émis immédiatement ou au travers du moteur d'émission.

send_packet La fonction send_packet permet l'envoi d'un paquet de données préalablement préparé par send_msg. Cette fonction est appelée par le moteur d'émission.

download_packet La fonction download_packet offre un accès aux paquets de données présents dans une zone mémoire gérée par le gestionnaire du périphérique de communication. Cette zone mémoire peut être le tampon de réception sur la périphérique de communication lui-même.

free_packet La fonction free_packet indique que le descripteur contenant le paquet en cours de réception peut être libéré.

is_full La fonction is_full indique si les tampons pour l'émission de paquets sont saturés. Ce test est employé par le moteur d'émission afin d'éviter la congestion de paquets à l'émission.

6.4 Modèle de protocoles de communication

La conception d'un protocole s'appuie sur l'ensemble des propriétés offertes par le modèle de matériel de communication. Chacune de ces propriétés peut être offerte à l'aide d'un support matériel ou entièrement simulée par voie logicielle (exemple de la diffusion sur un réseau point à point) selon les propriétés déclarées à l'initialisation du périphérique. Ainsi, le modèle de matériel se charge de l'acheminement des messages. Dans notre architecture, le protocole de communication ne doit définir que le moyen de créer un message lors de son émission, ainsi que la manière dont ce message est traité

lors de sa réception. Un protocole ainsi défini, s'intègre dans notre infrastructure de protocoles afin de fournir le service souhaité.

Dans la suite de ce chapitre, nous décrivons, dans un premier temps, les différentes parties constituant notre infrastructure de protocoles. Dans un second temps, nous détaillons les propriétés communes offertes par notre infrastructure de protocoles ainsi que les éléments à définir lors de la conception d'un nouveau protocole de communication.

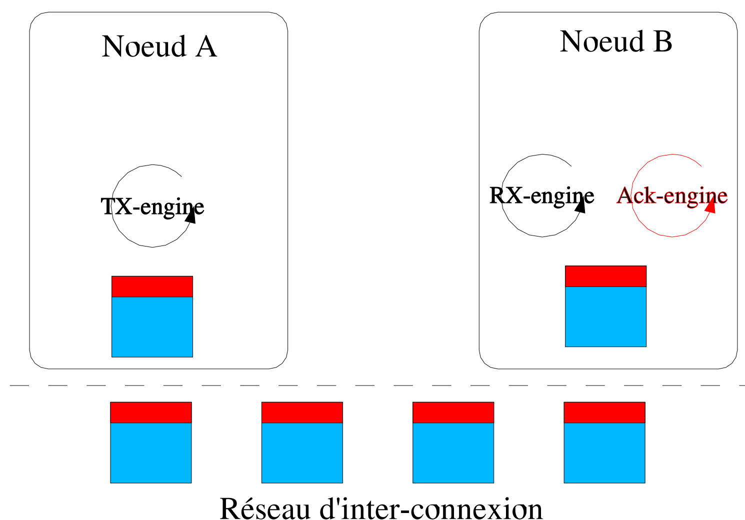


Fig. 6.2 – Moteurs de communication de Kerrighed

La figure 6.2 représente l'ensemble des unités de fonctionnement au sein de notre infrastructure de protocoles. Les propriétés offertes pour la conception de protocoles sont mises en œuvre dans ces unités. Les mécanismes d'envoi et de réception classiques emploient tout ou partie de ces moteurs.

Moteur d'émission Cette unité assure la gestion des files de paquets de données (créés par la primitive send_msg du matériel de communication) en attente d'émission. Ce même moteur assure les ré-émissions en cas de perte de paquet.

Moteur de réception Cette unité permet la reconstitution des messages à partir des paquets de données reçus. Ce moteur assure aussi la gestion de l'ordre des paquets et la détection d'éventuelles pertes de paquets.

Moteur d'acquittement Cette unité traite les acquittements embarqués dans les paquets de données reçus. En effet, le transport des acquittements est fait de manière traditionnelle par *PiggyBacking*. En cas de défaut d'acquittement, ce moteur déclenche auprès du moteur d'émission une ré-émission de la file de paquets.

6.5 Gestion des propriétés génériques dans l'infrastructure de protocoles

Bien que des protocoles distincts soient nécessaires pour les différents types de communication dans une grappe, certaines propriétés peuvent être communes et fournies comme base pour construire les protocoles. Ces propriétés ont généralement pour caractéristique d'être indépendantes des protocoles. Les propriétés communes sont des propriétés comme l'identification des nœuds, la gestion de la fragmentation des messages, les problèmes de retransmission de paquets et l'ordre des messages.

Afin d'exploiter cette architecture de protocoles, et de relier les messages aux protocoles leur correspondant, nous définissons la notion de **trame SSI**. Par la suite, une trame SSI est placée dans un paquet de données propre au périphérique de communication employé. La figure 6.3 détaille la structure d'un paquet de données. Un message est précédé d'informations liées à son protocole de communication. Cet ensemble de données est découpé en paquets de données à quoi est ajouté un en-tête générique.

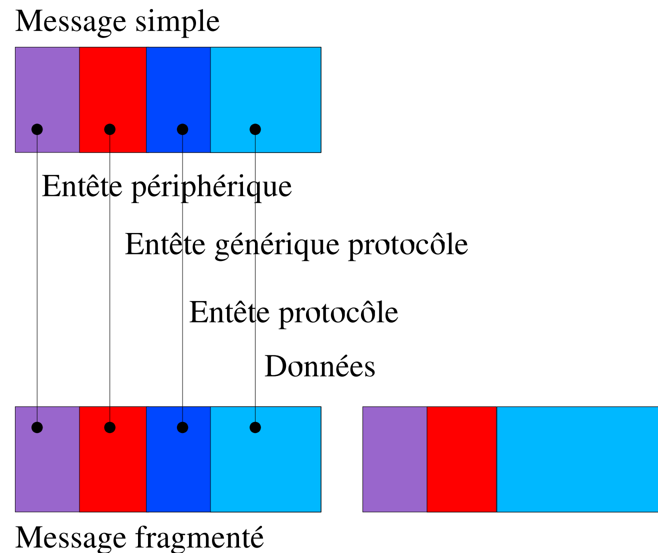


Fig. 6.3 – Trame SSI

6.5.1 Gestion de l'identification des nœuds

Lors de l'initialisation d'un nœud dans la grappe de calculateurs, un identifiant logique unique au sein de la grappe lui est attribué par l'infrastructure de protocoles. Cet identifiant, appelé **numéro de nœud** est interne au SSI et différent des identifiants matériels des périphériques de communication. Les identifiants matériels des périphériques de communication (tels que les adresses MAC) sont associés à ce numéro de nœud. La vocation première du numéro de nœud est de permettre la localisation de son correspondant lors d'un échange de messages. Ce numéro de nœud a une vocation

similaire au sein d'une grappe à celle du numéro de processeur au sein d'une machine multi processeur.

6.5.2 Gestion de la fragmentation

De nombreux matériels de communication imposent une taille de paquet maximale. Par exemple, la norme Ethernet prévoit un maximum de 1526 octets par trame. D'une manière générale, à chaque interface de communication réseau est associée une valeur représentant la taille de paquet (ou *Message Transfert Unit – MTU*).

Cependant, de nombreuses applications échangent des messages de taille bien largement supérieure aux MTU. Une fragmentation de ces messages est donc nécessaire. La fragmentation consiste à découper un grand message en plusieurs fragments de taille inférieure.

Étant donné le caractère dépendant du matériel de cette propriété, la fragmentation est réalisée au moment où le message est remis au gestionnaire de périphérique de la couche point-à-point. Après fragmentation, les paquets sont prêts à être transmis et enregistrés pour d'éventuelles retransmissions, de manière indépendante les uns des autres.

Chaque paquet ainsi généré dispose d'un **numéro de séquence**, rendant la paire {numéro de nœud du destinataire, numéro de séquence} unique dans le système.

À la réception, les messages sont ré-assemblés avant d'être remis au protocole adéquat.

6.5.3 Gestion des retransmissions

La fiabilité d'un système à image unique repose en grande partie sur la fiabilité de son réseau d'interconnexion. Dans le cas où le réseau physique n'offre pas de garantie concernant l'acheminement d'un paquet, un mécanisme spécifique doit être offert afin de permettre la retransmission de paquets perdus ou détectés corrompus. Pour cela, les paquets précédemment construits sont enregistrés dans une liste de ré-émission. Une alarme est associée à cette liste. Sur réception d'un paquet, le numéro de séquence est vérifié. En cas de perte ou de corruption de paquet, une demande explicite de ré-émission est faite à l'émetteur à l'aide du numéro de séquence. Pour ce mécanisme de retransmission, un algorithme classique de fenêtre d'acquittement est utilisé.

6.5.4 Gestion de l'ordre des messages

Disposer d'un système de communication respectant l'ordre d'émission des messages est une hypothèse communément admise dans les systèmes communicants. En effet, une telle hypothèse permet de simplifier grandement les automates d'état contrôlant un service distribué.

Pour établir un ordre des messages, chaque message est associé au numéro de séquence de son premier paquet de données. Ainsi la gestion de l'ordre des messages peut se déduire simplement à partir de l'ordre des fragments.

6.5.5 Modèle d'émission de message

Pour un protocole de communication, l'émission d'un message (figure 6.4) consiste principalement à créer l'en-tête de protocole générique et, le cas échéant, à fragmenter le message résultant. Les informations contenues dans l'en-tête de SSI générique contiennent notamment les numéros de séquence des paquets de données créés ainsi qu'un acquittement des paquets déjà reçus. Ces informations permettent au moteur de réception de gérer l'ordre des messages, de déclencher les ré-émissions de paquets, ainsi que de libérer les paquets acquittés.

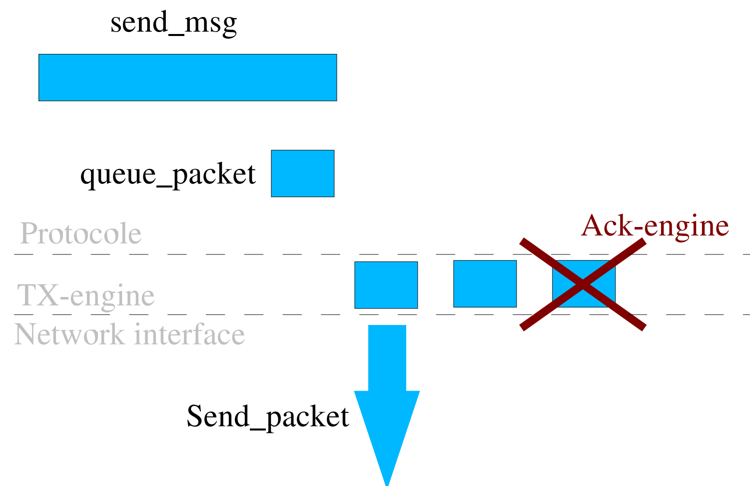


Fig. 6.4 – Émission d'un message

queue_packet Une fois les paquets de données créés par le gestionnaire du périphérique, ceux-ci sont remis au moteur d'émission. Ainsi le protocole peut avoir sa propre gestion de file d'émission et de ré-émission.

complete_header Permet la mise à jour, par le protocole, des paquets de données fournis par le gestionnaire de périphérique.

resume_send Lorsque l'envoi de nouveaux paquets est possible (après une phase de congestion par exemple), la fonction resume_send en informe le protocole.

sending_done La fonction sending_done acquitte l'envoi effectif d'un paquet de données au protocole. Cela permet notamment de libérer des ressources (mémoire) immobilisées par le paquet.

6.5.6 Modèle de réception de message

Notre approche vise à séparer la réception physique d'un message, du traitement qui va lui être associé. La réception physique est faite par le moteur de réception de message. Le traitement est effectué par le protocole de communication.

En réception, chaque message, chaque paquet de données ou chaque tampon de réception est attaché à un descripteur contenant les informations qui lui sont associées. Lorsque ces descripteurs sont créés par un protocole de communication, ils représentent un emplacement de réception de données et ils sont placés dans une liste correspondant aux réceptions en attente. Lorsque la création du descripteur fait suite à la réception d'un paquet provenant d'un périphérique, ce descripteur représente un paquet de données. Le but du moteur de réception est de rechercher le descripteur de réception correspondant au descripteur du fragment fraîchement reçu. Lorsqu'une correspondance est déterminée, le paquet reçu est enregistre a sa place définitive.

La figure 6.5 représente la réception d'un paquet par le moteur de réception ainsi que l'interaction de ce moteur avec le protocole de communication. Schématiquement, un paquet de données est réçu par le moteur de réception. Le couche de protocole fournit au moteur de réception, une liste de tampons de réception. Le moteur ré-assemble (éventuellement), ordonne les messages reçus et finalement les enregistre dans le prochain tampon, présent dans la liste. Le protocole effectue le traitement associé aux messages.

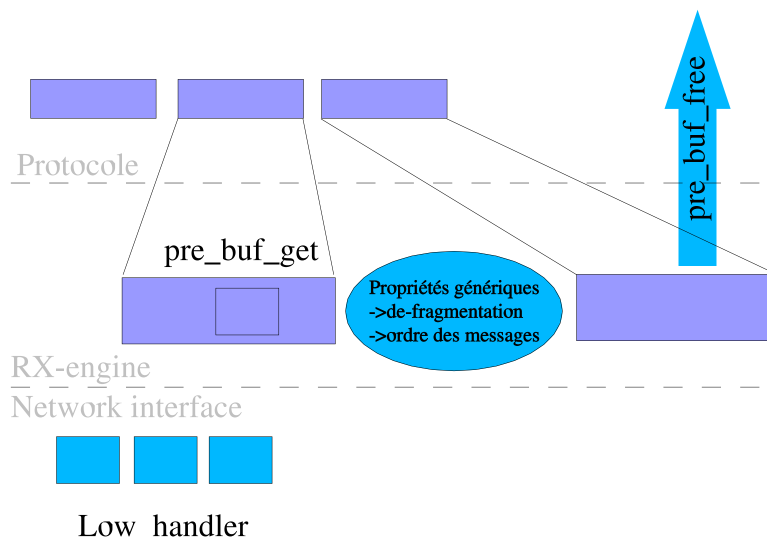


FIG. 6.5 – Réception d'un message

Au sein du moteur de réception, les descripteurs de réception changent de nombreuses fois d'état selon le type de tampon auquel ils sont associés. En particulier, l'état STATE_RECEIVING indique que le descripteur est associé à un message qui est en cours de réception. La principale conséquence de cet état est que le message associé

n'est pas cohérent (par exemple le message est fragmenté et tous les fragments n'ont pas été reçus).

low_handler constitue le point d'entrée du protocole pour la réception. Les paquets de données reçus par le périphérique de communication sont transmis à cette fonction pour être délivrés au service client du protocole. Ce client peut être un service système distribué ou une application communicante.

pre_buf_get À la réception d'un nouveau paquet de données, celui-ci est conservé dans une zone mémoire choisie par le matériel (zone tampon sur la carte mémoire). Afin de minimiser les recopies, le protocole peut fournir la méthode de réception des données et éventuellement fournir une zone tampon de réception. La fonction pre_buf_get fournit la méthode :

METHOD_PRE_BUF Le tampon est fourni par le protocole, les données sont directement placées dans cette zone mémoire. En fin de réception d'un message, le moteur de réception le transmet au protocole par l'appel pre_buf_free.

METHOD_POST_PRE_BUF Un tampon est fourni par le protocole, mais les données ne doivent pas y être placées immédiatement. La recopie est ultérieure et directement opérée par le protocole. En fin de réception du message, le moteur de réception le transmet au protocole par appel à la fonction pre_buf_free.

METHOD_UNEXPECTED Comme son nom le suggère, le fragment en cours de réception n'est pas attendu par le protocole. Par conséquent, celui-ci ne peut fournir aucun tampon. Le traitement du message se fait dans une zone mémoire gérée par le moteur de réception. En fin de réception du message, le protocole reçoit le message par la fonction unexpected_buffer_add.

METHOD_UNEXPECTED_NOACT Comme pour `METHOD_UNEXPECTED` le tampon n'est pas fourni par le protocole. Toutefois, le protocole décide de garder un lien vers ce nouveau descripteur et indique qu'aucun appel ultérieur au protocole ne sera nécessaire. Le protocole a la charge de vérifier l'état du descripteur associé (fin de l'état STATE_RECEIVING).

pre_buf_free La fonction pre_buf_free acquitte la réception d'un message dans le protocole pour un message reçu par la méthode METHOD_PRE_BUF. Cette fonction réalise le traitement qui est associé à ce message par le protocole.

unexpected_buffer_add La fonction UNEXPECTED_BUFFER_ADD tient un rôle symétrique à pre_buf_free pour un message reçu selon la méthode METHOD_UNEXPECTED.

Étudions le cas de la réception d'un long message dont le premier fragment n'est pas (encore) attendu. Ainsi, la réception du message débute en mode METHOD_UNEXPECTED. Toutefois, une demande de réception correspondant à ce message peut arriver avant que celui-ci ne soit complètement arrivé. Il est souhaitable de transférer la réception en cours vers sa nouvelle destination. La méthode METHOD_NEW_PRE_BUF correspond à cette situation. Pendant une réception de message, le protocole peut changer la méthode de réception pour METHOD_NEW_PRE_BUF. Cette méthode permet d'associer deux descripteurs de réception afin qu'ils partagent le même tampon de réception. Cette méthode provoque la recopie, par le moteur de réception, des données préalablement reçues dans l'ancien tampon vers le nouveau et change la méthode de réception en METHOD_PRE_BUF.

6.6 Résumé

Les besoins en communication d'un système à image unique sont variés. Il semble difficilement envisageable de concevoir un seul et unique protocole de communication répondant à toutes les contraintes spécifiées. Ce constat amène à la conception, non pas d'un protocole unique, mais d'une architecture permettant la réalisation des différents protocoles de communication souhaités.

L'architecture décrite permet de séparer la réception et le traitement des messages. La réception de message nécessite l'usage de techniques liées aux caractéristiques techniques du matériel ou à des propriétés génériques des protocoles de communication. L'architecture et ses propriétés génériques sont développées au travers d'un ensemble de moteurs configurables. Ainsi, les propriétés peuvent être choisies par le concepteur du protocole de communication. Le second aspect de notre architecture concerne le traitement des messages. Nous faisons le choix de laisser ce traitement au protocole de communication. Ainsi, le protocole de communication est à même de faire des choix tels que recopier directement les données dans la mémoire virtuelle d'une application ou initier l'exécution d'une procédure directement à partir de tampons de réception.

Finalement, nous avons décrit une interface logicielle pour les périphériques matériels afin de supporter efficacement les propriétés de performance souhaitées.

Chapitre 7

Système de communication pour la mise en place de services distribués dans un système d'exploitation pour grappe de calculateurs

La gestion globale des ressources au sein d'un système à image unique tel que Kerrighed nécessite la réalisation de services système distribués au sein de la grappe de calculateurs. De tels services distribués imposent l'usage d'outils de communication internes aux services, pour l'échange de données ou l'appel de procédures à distance. Dans un tel cadre, les communications se font entre les nœuds de la grappe (par opposition aux communications en processus déplaçables). Par ailleurs, les services distribués constituant le cœur de notre SE, nous devons en limiter les perturbations pour cause de reconfiguration de la grappe de calculateurs (suite aux ajouts, retraits de nœuds ou de liens de communication). Pour ce faire, nous avons choisi d'offrir la vision d'une grappe de calculateurs stable (et de taille maximale fixe) au-dessus d'un système de communication dynamique. Le système de communication détecte les changements de configuration et alerte l'architecture de services distribués qui a pour charge de masquer les changements.

La suite de ce chapitre est la présentation détaillée du concept de transaction de communication. Tout d'abord (paragraphe 7.1), en analysant les propriétés identifiées dans le chapitre 5, nous indiquons les principes sur lesquels notre architecture repose. Dans un second temps (paragraphes 7.2 et 7.3), nous présentons les concepts liés à la solution pour un système de communication dédié aux services systèmes distribués : les transactions de communication.

7.1 Cadre de l'étude

Nous nous intéressons, dans cette partie, à la conception d'un modèle de communication adapté à la conception et au déploiement de services système distribués pour grappe de calculateurs, tel qu'introduit dans les chapitres précédents. Notre démarche est de concevoir un système de communication adapté aux services (comme la gestion globale de la mémoire), et non pas d'adapter un tel service, complexe par nature, à un outil générique. En ce sens, les premières propriétés exigées par les clients sont :

– de donner l'illusion d'un accès exclusif au système de communication en multiplexant les communications (création de canaux de communication, virtuels)
– des communications fiables,
– des communications ordonnées.

Nous avons montré qu'un SE distribué repose sur un ensemble de services système distribués. Chacun de ces services est composé d'entités exécutées sur chacun des nœuds composant la grappe de calculateurs, et communiquant entre elles selon un modèle client/serveur. De l'efficacité à acheminer, délivrer et traiter ces requêtes vont dépendre les performances des services système distribués du SE distribué et par conséquent des applications de l'utilisateur. Le traitement de la requête est un problème interne au service système lui-même. En revanche, l'acheminement et la remise de cette requête concerne en premier lieu le système de communication.

Afin d'améliorer le parallélisme entre un client et le système de communication, le système Madeleine[11] a démontré qu'il fallait que le client ne fasse que décrire la communication désirée. Seul le système de communication est apte à prendre la décision optimale en fonction des demandes et des ressources réellement disponibles. Une telle approche peut être reprise en tenant compte du fait que l'ensemble de la mémoire d'un service système fait partie intégrante de la mémoire du système d'exploitation local. Par conséquent, dans un système Linux classique, cette mémoire bénéficie de propriétés telles qu'une adresse physique statique dans le temps et adressable par le matériel.

Le problème de déla remise de la requête est, quant à lui, solutionné par le principe des *Actives Messages*.

7.2 Concepts généraux liés aux transactions de communication

Afin de concevoir notre système de communication, nous sommes partis d'une constatation. La transmission d'une donnée (pour l'émetteur comme pour le récepteur) correspond à définir :

– un emplacement mémoire (adresse physique et taille de la donnée),
– une contrainte de communication (par exemple : usage immédiat ou différé),
– une action à réaliser à l'issue de l'opération de communication.

Les deux premières caractéristiques correspondent à ce que nous appelons la composition de message et permet de définir une requête de communication. La troisième caractéristique permet de définir une opération à réaliser au plus tôt après l'exécution

de la communication.

Définition 26 ***Élément de transaction de communication****: Nous appelons* ***élément de transaction de communication*** *l'association d'une zone de mémoire, d'une contrainte de communication (formant ainsi une requête de communication, à l'émission comme à la réception) et d'une action à réaliser.*

Nous rappelons que la mémoire d'un service système correspond à de la mémoire du SE lui-même. Dans le contexte du SE Linux (version 2.4), cette mémoire virtuelle est fixée définitivement à une plage de mémoire physique. De plus, cette mémoire est directement adressable pour des transferts en provenance ou à destination du matériel de communication.

Dans le cadre d'un élément de transaction, une action peut être associée aussi bien à l'émission qu'à la réception de données. Dans le cadre de l'émission, une telle action peut permettre de libérer des ressources réservées le temps de la communication (zone mémoire, jeton de contrôle de flux, etc.). Le cadre de la réception correspond tout naturellement à celui du traitement de la requête fraîchement reçue.

Un élément de transaction permet de définir une opération de communication atomique. Toutefois, dans bon nombre de cas, la communication voulue va contenir différentes informations. Au lieu de recopier ces données dans un seul et même message, nous souhaitons définir autant d'éléments de transaction qu'il y a de données. Toutefois, ces données formant un ensemble cohérent pour la requête demandée, il est nécessaire de disposer d'un outil permettant de ré-assembler (logiquement) les différents fragments composant la requête émise.

Définition 27 ***Transaction de communication****: Nous appelons* ***transaction de communication*** *un ensemble d'éléments de transaction faisant partie de la même unité d'action logique.*

Grâce à la notion de transaction de communication, il est possible de décrire le message de la transmission. En associant à chaque élément de transaction une action, il est possible de décrire les actions à réaliser au fur et à mesure que la requête est réceptionnée. À l'instar de la composition de message, nous pouvons parler de composition d'action pour la requête.

L'ensemble des actions associées aux éléments de transaction forme un traitement associé à la communication. Ce traitement n'est pas, à l'instar des processus, manipulé par l'ordonnanceur, mais au contraire dépend d'événements de communication. Il est donc concevable qu'une action en début de transaction souhaite transmettre des informations à une action ultérieure.

Définition 28 ***Contexte de transaction****: Nous appelons* ***contexte de transaction*** *un espace de données partagées accessibles à tous les éléments d'une transaction.*

À l'aide de ce contexte de transaction, il est possible de définir un modèle d'exécution uniquement fondé sur les événements du système de communication.

7.3 Les transactions de communication

L'objectif des transactions de communication est de permettre une description efficace d'un modèle d'exécution des appels de procédure à distance. Dans ce paragraphe, nous décrirons précisement les concepts précédemment énoncés et nous indiquons les problèmes résolus.

7.3.1 Transaction de communication

Une transaction de communication permet de créer un lien entre une succession d'éléments de transaction. En terme de message, la transaction de communication peut être assimilée à un message alors que chaque élément de transaction ne constitue qu'un fragment de ce message.

Multiplexage des commmunications Afin de multiplexer l'accès à la ressource de communication, les transactions disposent de plusieurs attributs globaux à la transaction. Ces attributs sont :
- le nœud interlocuteur,
- un identifiant de canal,
- un identifiant de sous-canal.

Définition 29 ***Circuit de communication:*** *Nous appelons* ***circuit de communication*** *l'abstraction correspondant à la désignation des messages en provenance d'un nœud, d'un canal et d'un sous-canal déterminés.*

Les figures 7.1 et 7.2 présentent respectivement l'interface retenue pour déclarer une transaction pour l'émetteur et le récepteur. Nous y retrouvons les déclarations des caractéristiques globales à la transaction. En retour de la création d'une transaction, un descripteur est fourni au client afin de pouvoir réaliser la composition de son message et les actions associées.

Listing 7.1 – Interface de transaction pour les émissions

```
struct gimli_desc* gimli_begin_pack(kerrighed_node_t dest,
                                    gimli_channel_t channel,
                                    gimli_schannel_t schannel,
                                    gimli_transact_cb_t cb);
int gimli_end_pack_with_callback(struct gimli_desc* desc,
                                 int flags, gimli_transact_cb_t cb);
int gimli_end_pack(struct gimli_desc* desc);
```

Listing 7.2 – Interface de transaction pour les réceptions

```
struct gimli_desc* gimli_begin_unpack(kerrighed_node_t sender,
                                      gimli_channel_t channel,
                                      gimli_schannel_t schannel);
int gimli_end_unpack_with_callback(struct gimli_desc* desc,
                                   int flags, gimli_transact_cb_t cb);
int gimli_end_unpack(struct gimli_desc* desc);
```

Le descripteur (figure 7.3) est une structure opaque pour le client. Il permet au système de communication la mise en œuvre du protocole associé à ces transactions.

Listing 7.3 – Descripteur de communication

```
struct gimli_desc {
    ...
    kerrighed_node_t node;
    gimli_channel_t channel;
    gimli_schannel_t schannel;
    gimli_group_t group;
    ...
    void* private_data;
    gimli_transact_cb_t cb, end_cb;
};
```

Notion d'ordre dans les transactions Par défaut, la notion d'ordre n'est valable qu'au sein d'un circuit de communication. Seuls les débuts de transaction sont acheminés et délivrés de manière ordonnée. Par la suite, les éléments de transaction ne sont ordonnés qu'au sein de leur transaction. Ainsi, deux transactions entrelacées sur le même circuit de communication ne peuvent pas se pénaliser.

Synchronisation des transactions Lorsque plusieurs éléments de transaction ont été posés, il peut être nécessaire de s'assurer que l'un de ces éléments de transaction a bien été réalisée avant de poursuivre son exécution. Les fonctions gimli_wait_pack et gimli_wait_unpack permettent d'attendre l'exécution d'un élément de transaction précédemment posé.

Sérialisation des transactions Toutefois, il existe de nombreuses situations où le concepteur de services distribués désire avoir une maîtrise plus fine de l'ordre de réception et surtout du traitement des transactions. En particulier lors du déplacement de ressources système, il est nécessaire de disposer d'un ordre séquentiel strict au sein d'un ou plusieurs circuits de communication. Un mécanisme de sérialisation est donc nécessaire.

Notre mécanisme de sérialisation est fondé sur l'attribution d'un **jeton d'action** à un ensemble de canaux (appelé **groupe de canaux**). Lors d'une réception sur un canal associé à un jeton d'action, celui-ci est implicitement fourni au client. En fin de traitement de la requête, le jeton doit être restitué à son groupe. Ainsi, le système de communication peut procéder à la remise de la requête suivante.

La création d'un jeton d'action se fait implicitement par la création d'un groupe de canaux (figure 7.4 – gimli_group_create). Les canaux sont associés à un jeton par ajout dans le groupe associé (gimli_group_add). En fin de vie d'un groupe, un canal est retiré du groupe par gimli_group_del, et le groupe (ainsi que le jeton d'action) sont détruits par gimli_group_destroy.

Listing 7.4 – Interface de gestion des groupes de canaux

```
gimli_group_create
```

```
gimli_group_destroy

gimli_group_add
gimli_group_del
```

Les fonctions gimli_group_serialize et gimli_group_unserialize (figure 7.5) activent ou désactivent l'usage du jeton d'action. Enfin, la fonction gimli_group_next_receive restitue le jeton au système de communication. Dans le cas particulier des transactions, la sérialisation implique notamment que la transaction courante soit terminée avant que la suivante ne puisse débuter.

Listing 7.5 – Interface de gestion du jeton d'action

```
gimli_group_serialize
gimli_group_unserialize

gimli_group_next_recveive
```

Annulation d'une transaction Communément, les transactions en réception sont supposées symétriques des transactions en émission. Cela implique que les requêtes de communication soient dans le même ordre et portent sur le même nombre de tampons de même taille. En pratique, il peut être souhaitable d'interrompre une transaction en réception, lorsque pour une raison quelconque, le récepteur ne souhaite pas réaliser les traitements liés à une transaction. La fonction gimli_cancel_unpack (figure 7.6) permet de ne pas poursuivre la remise des données : les messages sont effectivement reçus mais immédiatement détruits.

Listing 7.6 – Abandon d'une transaction

```
int gimli_cancel_unpack(struct gimli_desc* desc);
```

7.3.2 Élément de transaction

L'élément de transaction constitue le cœur d'une transaction. Il permet de poster une demande d'émission ou de réception au système de communication. Cette demande est composée d'un emplacement mémoire, d'une contrainte ainsi que d'une action à réaliser au plus tôt après la réalisation de l'opération de communication.

Listing 7.7 – Interface de transaction pour les emissions

```
int gimli_pack_with_callback(struct gimli_desc* desc, int flags,
                             void *buffer, int size, gimli_transact_cb_t cb);
int gimli_pack(struct gimli_desc* desc, int flags, void *buffer, int size);


int gimli_unpack_with_callback(struct gimli_desc* desc, int flags,
                               void *buffer, int size,
                               gimli_transact_cb_t cb);
int gimli_unpack(struct gimli_desc* desc, int flags, void *buffer, int size);
```

Contraintes de communication Lorsque le client soumet au système de communication une zone de sa mémoire, des problèmes d'accès concurrents peuvent survenir. Afin de privilégier les recouvrements d'exécution entre le client et le système de communication, un mode non bloquant des opérations est réalisé par défaut. Ainsi, le système de communication a entière liberté pour réaliser le transfert des données en une ou plusieurs trames sur le réseau physique. Dans ce cas précis, il n'est pas possible de déterminer le moment exact où la création de la trame est effective. Un problème analogue existe lors de la réception de messages. Le contrat d'utilisation des transactions implique de ne pas accéder à une mémoire confiée au système de communication, sous peine de comportement non déterminé. Afin de mieux contrôler ce paramètre, un ensemble de contraintes similaires à celles de Madeleine a été introduit.

Les contraintes de communications sont :

- **GIMLI_TRANSACT_WAIT** : force l'attente du traitement de la requête,
- **GIMLI_TRANSACT_SAFE** : force une recopie du tampon de communication,
- **GIMLI_TRANSACT_LATER** : réserve l'emplacement dans le message mais réalise le transfert effectif en fin de transaction,
- **GIMLI_TRANSACT_IMMEDIATE** : demande l'envoi (effectif) du tampon au plus tôt,
- **GIMLI_TRANSACT_CHANGING** : signale que le tampon peut-être modifié avant la fin de la transaction.

Actions associées à une requête de communication Jusqu'à présent, nous n'avons décrit que l'aspect de composition de message d'une transaction. Toutefois, par l'intermédiaire des actions associées à chaque élément de transaction, il est possible de définir un fil d'exécution conduit selon les événements du système de communication. Une action est réalisée à la suite de la réception d'un fragment de message et dans le même contexte que celui-ci. Ainsi, pour une machine Linux dotée d'une carte Ethernet, l'action associée sera exécutée dans le contexte d'un fil d'exécution en interruption. Une telle propriété garantit que le traitement associé au fragment de message sera effectué au plus tôt après réception des données. En contre-partie, un tel traitement se doit d'être suffisament léger pour ne pas pénaliser le reste du système.

7.3.3 Contexte de transaction

Au sein d'une transaction, les opérations réalisées par les éléments de transaction sont généralement liées. Ainsi, un état est créé puis modifié lors des exécutions successives de ces éléments. Dans le cadre d'un processus, un tel état serait enregistré dans une portion de la mémoire du processus, faisant ainsi partie de son contexte.

Un tel contexte n'existant pas pour les fils d'exécution en interruption, nous devons introduire la notion de contexte de transaction.

Définition 30 ***Contexte de transaction:*** *Nous appelons* ***contexte de transaction*** *une zone de mémoire commune à l'ensemble des éléments d'une transaction et accessible par les actions associées.*

Un tel contexte de transaction peut avoir une durée de vie limitée à la transaction elle-même (appelé **contexte neutre**) ou au contraire être un objet persistant dans le système (appelé **contexte objet**). Le contexte est référencé par le descripteur de transaction (champ : private_data).

Contexte neutre Le contexte neutre correspond à un contexte uniquement créé pour le traitement de la transaction. L'allocation et l'initialisation de ce contexte se font au moment de la création de la transaction. Un tel contexte permet à un élément de transaction de laisser un état pour les éléments de transaction suivants.

Contexte objet Le contexte objet permet d'associer les traitements à un objet système.

Un système d'exploitation doit prendre en compte la gestion des ressources du système. Ces ressources peuvent correspondre à des ressources physiques telles que les pages mémoire et les blocs disque. Ces ressources physiques sont virtualisées par le système d'exploitation et offertes aux applications sous forme de ressources logiques. De plus, des ressources purement logiques (sans support direct par un matériel quelconque) telles que les verrous système et les *sockets* existent et doivent être prises en charge. Afin d'assurer le bon usage de ces ressources logiques, le système d'exploitation associe à chacune des ressources un état interne représentant la ressource ainsi qu'une liste des opérations autorisées pour cette ressource. Chacune de ces opérations correspond à un type de transaction possible sur l'objet. Ces opérations influencent directement l'état interne de la ressource. Pour une ressource simple telle qu'un verrou système, l'état interne correspond à l'état du verrou (pris ou libre) et les actions sont par exemple : *prendre le verrou* et *libérer le verrou.*

Une ressource logique se caractérise donc par son état interne et les actions qu'elle supporte.

7.4 Résumé

Les services système distribués sont conçus sur un modèle client/serveur à base de requêtes. Des performances dans l'acheminement et le traitement de ces requêtes dépendent les performances globales de la grappe de calculateurs. Une requête est généralement créée à partir de données collectées sur le nœud émetteur. Cette requête est ensuite transmise au nœud récepteur qui en débute l'analyse, et active le traitement pour accomplir la tâche demandée.

Dans ce chapitre, nous avons présenté un modèle de programmation et d'exécution d'un mécanisme d'appel de fonction à distance appelé transaction de communication. Les transactions de communication permettent de décrire à la fois le contenu du message et une série d'opérations à exécuter en parallèle de l'émission ou de la réception de la requête. Ces opérations sont réalisées au plus tôt après le traitement de la requête. Généralement, ce traitement est effectué dans le cadre d'un fil d'exécution en interruption.

Afin de lier le traitement des fragments de message entre eux, les transactions disposent d'un modèle de contexte permettant de réaliser la tâche souhaitée en plusieurs étapes. Ces contextes peuvent être de deux types : soit temporaires et ils disparaissent en même temps que se termine la transaction, soit persistants et ils ont une existance indépendante des transactions. Les contextes persistants (ou contextes objet) définissent un modèle d'objets système auxquels sont associées des méthodes pour les manipuler. Ces méthodes sont représentées par les transactions de communication.

Chapitre 8

Flux de données dynamiques

Le chapitre précédent a présenté un modèle de communication adapté à la programmation de services système distribués. Le concept de transaction de communication s'adresse à des entités faisant partie du SE local de chacun des nœuds de la grappe. De ce fait, il est évident qu'il n'aide en rien au déplacement efficace de processus communicants intégrés dans une application parallèle fondée sur le modèle de communication par échange de messages.

Dans un SSI tel que Kerrighed, l'état interne d'un processus doit pouvoir être capturé afin de déplacer ou de réaliser un point de reprise de ce processus. Le cas particulier des processus communicants oblige à devoir capturer, en plus de l'état du processus, les états des canaux de communication en cours d'utilisation par ces processus.

Ce chapitre traite du support nécessaire pour le déplacement ou le calcul de points de reprise de processus communiquant par message. Dans une première partie (paragraphe 8.1), nous présentons quelques interfaces de communication existantes, et nous caractérisons les flux de données qui y sont liés. Dans la suite de ce chapitre, nous nous intéressons tout d'abord aux communications *intra-grappes*. Dans le paragraphe 8.4, nous discutons des communications *inter grappe* entre deux grappes exploitant un même SSI. Enfin, le paragraphe 8.3 est consacré aux communications *extra-grappes* entre une machine quelconque (avec un SE quelconque) et une grappe disposant d'un SSI.

8.1 Notion de flux de données

Définition 31 ***Flux de données:*** *Un* ***flux de données*** *est un mécanisme permettant de transmettre une suite d'octets entre deux entités distinctes appelées extrémités.*

Un grand nombre d'applications, susceptibles d'être exécutées sur une grappe de calculateurs, emploie des flux de données pour échanger des informations entre des processus. Les systèmes d'exploitation de la famille Unix offrent différents types de flux. Chacun de ces types de flux correspond à un problème ou une philosophie particulière. Ainsi les pipe et fifo correspondent à des mécanismes d'échange de données à l'intérieur d'une même machine. L'interface socket est une interface d'avantage orientée système

de communication. Les sockets ont été déclinées dans diverses familles, notamment INET pour les communications entre machines et UNIX pour les communications internes à une machine.

La notion de flux de données intègre selon nous, trois aspects :

- le transfert d'une suite d'octets entre deux ou plusieurs extrémités d'un flux,
- le protocole régissant l'identification, l'accès et le fonctionnement du flux,
- la localisation des extrémités.

L'échange d'octets est la définition même du flux. Nous souhaitons mettre à disposition, le plus efficacement possible, des octets présents en un point `A` sur un point `B`. Quelle que soit l'interface de communication retenue, cette opération élémentaire doit être réalisée. Le protocole de communication assure la mise en place des flux et permet de définir les propriétés attachées aux flux. Ce protocole est fortement lié au type d'interface de communication souhaité. Enfin, la localisation des extrémités est le point sensible dans un SSI où les processus peuvent être déplacés par un ordonnanceur global mettant en œuvre une politique d'équilibrage de charge dynamique.

Nous distinguons trois catégories de flux (voir paragraphe 5.3) selon la localisation des processus situés aux extrémités du flux. Dans une communication *intra-grappe*, les processus communicants appartiennent à la même grappe de calculateurs. Une application écrite selon le modèle de programmation MPI et déployée sur une seule grappe correspond à ce type de communication. Si cette même application est déployée sur deux grappes de calculateurs différentes (mais utilisant le même SSI), nous parlons de communication *inter-grappe*. Enfin, le cas où l'application est déployée sur deux grappes distinctes ne partageant pas le même système d'exploitation correspond aux communications *extra-grappe*.

Dans le cadre d'un système à image unique, l'usage des flux doit être transparent à la localisation et aux déplacements des processus. En effet, suite à une décision de l'ordonnanceur global, un processus peut être déplacé à tout moment de son exécution : les extrémités de flux associées doivent être déplacées, elles aussi. Cette propriété de transparence à la localisation est équivalente à la propriété de transparence vis-à-vis du processeur dans une machine SMP. Une solution naïve pour assurer la transparence de la localisation consiste à utiliser la notion de *home node* (voir figure 4.1). Afin de masquer le déplacement d'un processus, un processus intermédiaire est créé pour relayer l'ensemble des communications. Clairement, l'ajout de cet intermédiaire pénalise les performances du flux de communication après migration d'une extrémité et ceci en raison de l'indirection par un nœud supplémentaire.

Notre objectif est de concevoir un système de communication permettant l'exécution efficace d'applications communiquant par flux malgré le déplacement (ou le redémarrage à partir d'un point de reprise) d'un ou plusieurs processus de cette application en cours d'exécution. Nous souhaitons que notre architecture puisse s'adapter au plus large ensemble d'interfaces de communication et donc de types de flux possible. Idéalement, une application communiquant par flux doit pouvoir être déployée sur notre architecture (et bénéficier de ses propriétés) sans modification de son code source, recompilation ou même nouvelle édition de lien.

8.2 Communication intra-grappe de calculateurs

Déployer une application communiquant par message sur un système à image unique tout en bénéficiant des propriétés de placement, déplacement et calcul de point de reprise des processus, nécessite un support adapté de la part du système de communication. L'étude de ce problème est simplifié par le fait que ces communications restent locales au système de communication du SSI. Sous réserve de respecter les interfaces de programmation, il n'y a donc aucune obligation de respecter les protocoles de communication tels qu'ils sont décrits dans les manuels. En contre-partie, une interface standard et une interface modifiée ne pourront communiquer par le même flux de donnée.

8.2.1 Architecture pour la haute performance

Les systèmes de communication par indirection des messages permettent de conserver une véritable compatibilité avec les protocoles standard de communication. Dans le cas de communication dans lesquelles l'ensemble des extrémités se situent au sein de la même grappe, l'avantage de la compatibilité devient la principale source de dégradation des performances.

L'objectif est de préserver des latences et des bandes passantes de communication comparables avant et après le déplacement d'un processus. Le seul moyen d'y parvenir est de maintenir une communication directe avant et après le déplacement du processus.

Afin de maintenir une communication directe au sein d'un flux, un mécanisme de localisation permanente des extrémités du flux doit être conçu. La contre-partie des communications directes est un protocole de déplacement de processus devant assurer la mise à jour de la localisation des extrémités.

8.2.2 Introduction aux flux dynamiques

Un mécanisme de communication par message peut être décomposé en un protocole lié à l'interface de communication et en un flux de données indépendant du protocole. Nous identifions ainsi une partie générique : le flux de données, et une partie spécifique : le protocole de communication. Afin de conserver la transparence de nos mécanismes de communication, les interfaces de communication ne doivent pas être modifiées.

Le déplacement de processus est une opération coûteuse pour l'application. Aussi l'une des tâches de l'ordonnanceur global de processus est de limiter les "ping-pong" de processus entre les nœuds. Nous pouvons donc faire l'hypothèse que le déplacement de processus n'est que ponctuel dans l'exécution d'un processus. Ainsi, notre architecture a pour objectif de permettre des échanges de message sans surcoût lié à la propriété de processus mobiles. En contre-partie de ce choix, nous reportons toute la complexité de cette propriété dans les phases de mise en place et de reconfiguration (par exemple lors d'un déplacement d'un processus) des flux.

Définition 32 ***Flux dynamique****: Nous appelons* ***flux dynamique*** *un mécanisme assurant des échanges de messages entre des* ***extrémités*** *mobiles.*

Les extrémités d'un flux sont les interfaces de communication et de contrôle de ce flux.

8.2.2.1 Intégration des flux dynamiques au sein des mécanismes de communication standard

Les applications que nous souhaitons déployer sur notre système à image unique communiquent par des interfaces établies de longue date dans le domaine des systèmes d'exploitation. Ces interfaces ne prévoient pas le support de propriétés telles que le déplacement, et bien entendu ne prévoient pas l'usage de flux dynamiques. Par conséquent, il est nécessaire d'insérer notre mécanisme de communication au sein de ces interfaces déjà établies. Toutefois, chaque interface de communication dispose de son propre protocole et par conséquent il n'est pas envisageable d'intégrer, dans les flux dynamiques, l'ensemble des protocoles existants. Une approche laissant la gestion des protocoles en dehors de la notion de flux dynamiques est préférable.

Définition 33 ***Interface de protocole****: Nous nommons* ***interface de protocole****, l'interface de communication classique étendue par l'usage des flux dynamiques.*

Cette interface de protocole a pour but :
- d'offrir une interface compatible avec un standard donné en dehors des phases de déplacement des processus communicants,
- de masquer le protocole de déplacement d'une extrémité,
- d'éventuellement étendre le protocole standard pour permettre la mise en œuvre d'optimisations liées aux flux dynamiques.

Afin de réaliser une interface de protocole, seuls les deux premiers points sont nécessaires. Dans tous les cas, l'extension du protocole doit rester optionnelle. Le bon fonctionnement d'une application quelconque avec le SSI doit se faire en toute transparence à l'aide des interfaces de programmation habituelles. Une éventuelle extension du protocole ne peut être envisagée que dans le contexte d'une amélioration des performances dans le cadre d'une application conçue spécialement pour notre architecture.

8.2.2.2 Caractérisation des extrémités d'un flux dynamique

Afin d'être opérationnel, un flux dynamique nécessite d'être créé et d'identifier ses extrémités. Il existe donc un état interne à un flux dynamique.

Définition 34 ***État d'un flux****: Nous appelons* ***état d'un flux*** *les propriétés d'un flux permettant :*
- *d'identifier le flux,*
- *de déterminer si le flux est actif ou pas (les données peuvent-elles circuler ou pas ?),*
- *d'obtenir la liste des extrémités enregistrées dans le flux.*

Par conséquent, au sein d'un flux dynamique deux fonctionnalités distinctes cohabitent. D'une part, les flux doivent permettre un échange de données entre les extrémités. D'autre part, l'état interne d'un flux dynamique doit pouvoir être modifié dans le temps. Pour un flux donné, l'état de ce flux est localisé sur un nœud qui en assure la gestion.

Définition 35 ***Gestionnaire de flux****: Nous appelons* ***gestionnaire d'un flux*** *le nœud assurant la gestion de l'état du flux concerné.*

Le gestionnaire d'un flux est un nœud quelconque. Notamment, il n'est pas nécessaire qu'un processus attaché à ce flux soit exécuté sur le nœud gestionnaire.

Définition 36 ***Socket d'E/S****: Nous appelons* ***socket d'E/S*** *une extrémité permettant d'effectuer uniquement des opérations d'envoi et/ou de réception de messages.*

Définition 37 ***Socket d'interface****: Nous appelons* ***socket d'interface****, une extrémité permettant d'effectuer l'ensemble des opérations de contrôle d'un flux sauf les opérations d'entrée ou de sortie.*

Par conséquent, nous pouvons définir une extrémité comme étant un ensemble de *sockets d'E/S* ainsi que de *sockets d'interface*. Le nombre de sockets d'E/S ainsi que de sockets d'interface n'est pas limité. En effet, les sockets d'E/S et les sockets d'interface ne sont que des points d'accès à l'extrémité du flux. Si nous nous replaçons dans un SE classique, cette situation est comparable au fait que plusieurs descripteurs de fichiers peuvent pointer sur la même interface de communication.

Concernant une socket quelconque, sa propriété essentielle est sa localisation. Ainsi nous définissons deux propriétés relatives à la localisation d'une socket.

Propriété 12 ***Socket attachée à un flux****: Nous disons qu'une socket est* ***attachée*** *à un flux lorsque le gestionnaire du flux dispose de la localisation exacte de cette socket*

Propriété 13 ***Socket détachée d'un flux****: Nous disons qu'une socket est* ***détachée*** *du flux, lorsque le gestionnaire ne dispose pas d'information sur la localisation de cette socket*

Définition 38 ***Extrémité distribuée****: Nous appelons* ***extrémité distribuée****, une extrémité de flux dont plusieurs instances existent sur des nœuds distincts.*

Classiquement, afin de manipuler une interface de communication standard, le processus détient une référence sur l'interface de communication au travers d'un descripteur (de fichiers dans les systèmes Unix). À la suite d'opérations système particulières (telles que le clonage de processus ou la création de processus légers), une même interface peut être référencée par plusieurs descripteurs appartenant à différents processus. Au sein d'un système à image unique, nous devons donc envisager la possibilité qu'une même extrémité puisse être partagée entre différents processus s'exécutant sur différents nœuds.

L'aspect distribué de l'extrémité provient de la localisation sur différents nœuds de ces sockets. Par ailleurs, nous pouvons définir un flux dynamique comme étant un ensemble d'extrémités (et donc de sockets). Dans ce cas, le nombre d'extrémités présentes dans le flux dépend de l'interface de communication que l'on souhaite réaliser. Ainsi, un flux mettant en œuvre une interface de type pipe non nommé contiendra deux extrémités distribuées.

Le concept d'extrémité distribuée permet d'offrir un niveau de partage équivalent à celui d'une structure en mémoire partagée. Toutefois, l'aspect distribué d'une extrémité créé un problème pour l'échange de données. En effet, si une même extrémité distribuée est instanciée sur deux nœuds différents, le problème de l'acheminement direct des messages du processus émetteur vers le processus récepteur doit être résolu.

8.2.2.3 Synthèse des concepts liés aux flux dynamiques

La figure 8.1 illustre, pour le cas des sockets, les rôles séparés des sockets d'E/S, des sockets d'interface ainsi que des interfaces de protocole. Tout d'abord, nous rappelons brièvement quelques éléments de l'interface de programmation socket. Le modèle de programmmation des sockets suit un modèle de type send/recv auquel est associé un protocole pour l'identification d'un interlocuteur et l'établissement d'une communication avec celui-ci. Les sockets constituent une des interfaces standard dans les SE de type Unix et sont mises à disposition des applications sous forme d'appels système.

En utilisant une interface de communication standard, un processus communiquant par sockets au sein de notre architecture de communication utilise en réalité trois interfaces différentes selon le type d'opérations réalisées. Les opérations strictement protocolaires (telles que socket et ioctl) sont réalisées par une interface de protocole. Lors de la création effective d'un canal de communication, en vue d'un véritable échange de données, une socket d'interface est créée. Enfin lors des opérations d'envoi et de réception de données, une socket d'E/S est définie. La gestion de ces trois interfaces est réalisée en modifiant les primitives système de l'interface socket.

La figure 5.1 représente la pile réseau de notre système à image unique telle qu'utilisée par les applications. Au niveau le plus bas, se trouve le modèle de matériel de communication (paragraphe 6.3). Cette interface est exploitée par un protocole de communication dédié au flux de communication, au sein de l'infrastructure des protocoles de communication (paragraphe 6.4). Ce protocole de communication permet l'échange de messages hors migration (support aux sockets d'E/S). La gestion des flux dynamiques et le support aux *sockets d'interface* réalisés dans un service distribué dédié (appelé *Palantir*). Enfin les *interfaces de protocole* sont fournies au sein de services distribués supplémentaires.

La suite de cette partie est consacrée à détailler le concept de flux dynamique. Dans un premier temps, nous présentons (paragraphe 8.2.3) l'architecture permettant de réaliser des communications directes entre les processus. Dans le paragraphe 8.2.4, nous présentons le protocole permettant de localiser une extrémité de communication lors de ses déplacements. Puis nous présentons (paragraphe 8.2.5) l'intégration de nos mécanismes au sein de deux modèles de communication. Enfin, le paragraphe 8.2.6

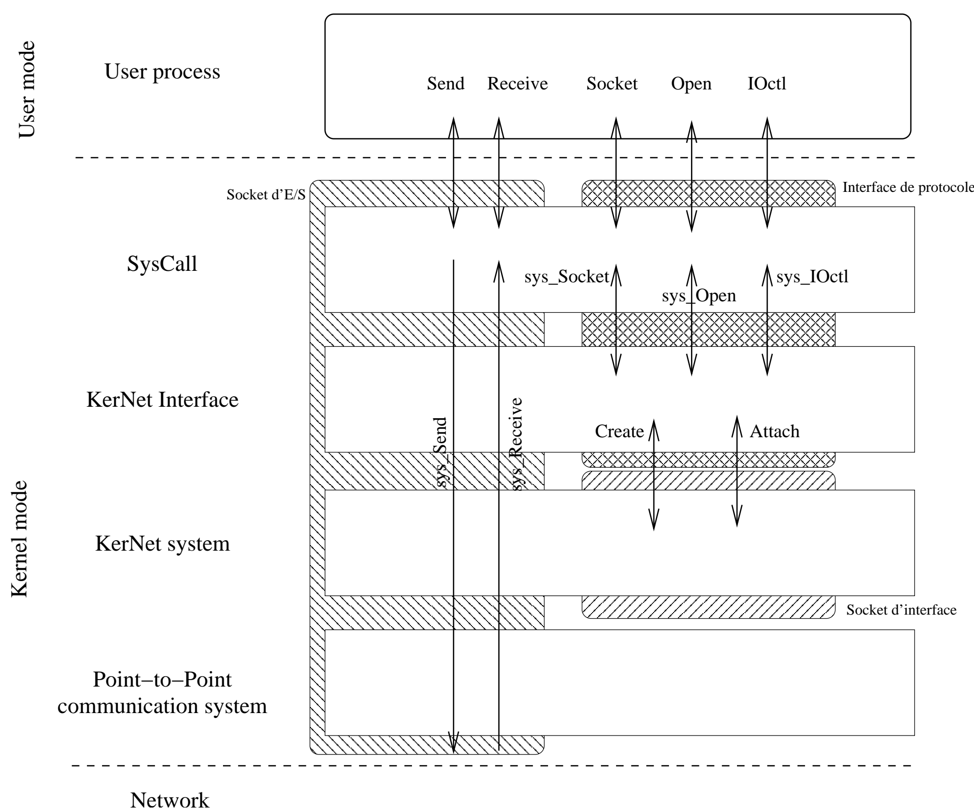


FIG. 8.1 – Exemple de l'architecture des flux dynamique pour la mise en œuvre de sockets

détaille le support que les flux dynamiques offrent aux mécanismes réalisant des points de reprise d'application.

8.2.3 Communication directe entre les extrémités distribuées

Mécanisme général de communication Un flux dynamique est composé de plusieurs extrémités de communication. Pour simplifier notre propos, nous supposons que chacune des extrémités n'est composée que d'une socket d'E/S. Les données échangées au sein d'un flux dynamique sont acheminées d'une socket d'E/S attachée au processus émetteur vers une autre socket d'E/S attachée au processus récepteur. Afin de transmettre les données sans indirection, l'émetteur d'un message doit donc connaître l'adresse (*ie.* le numéro de nœud) de la socket d'E/S du destinataire.

Afin de localiser les sockets d'E/S, le flux dynamique tient à jour la liste des localisations des sockets d'E/S disponibles pour chaque extrémité. Cette liste est, par la suite, propagée à chacune des sockets d'E/S.

Définition 39 ***Carte des extrémités d'un flux (définition partielle):*** *La* ***Carte d'extrémités*** *pour une socket d'E/S est la liste des localisations des autres sockets*

d'E/S du flux.

Concurrence au sein d'une extrémité de communication Une extrémité distribuée peut être composée de plusieurs *sockets* réparties sur plusieurs nœuds. Ces *sockets* peuvent être de deux types : sockets d'E/S ou *sockets d'interfaces*. Par définition, seules les *sockets d'E/S* communiquent, par conséquent elles seules doivent être prises en compte pour établir une communication directe entre les processus communicants. Dans le paragraphe précédent, nous avons restreint notre étude au cas particulier d'une extrémité ne comprenant qu'une seule *socket d'E/S*. Dans le cas général, plusieurs *sockets d'E/S* peuvent faire partie d'une même extrémité de flux. Cette situation se produit notamment lorsque l'interface de communication associée à l'extrémité est dupliquée (cas de l'appel système fork qui duplique un processus). Dans ce cas, afin d'acheminer les messages directement à leur destinataire, il est nécessaire d'identifier la "bonne" *socket d'E/S*.

Définition 40 ***Jeton de communication:*** *Le* ***jeton de communication*** *est un marqueur unique pour une extrémité distribuée signalant la socket d'E/S autorisée à réaliser une action d'émission ou de réception.*

Le jeton de communication est nécessaire avant toute opération de communication car il permet aux autres extrémités d'identifier précisément leur correspondant. La gestion du jeton se fait à l'aide de deux primitives (voir *listing* 8.1). La primitive principale kernet_req_io attribue le jeton de communication au processus appelant. Ainsi, la liste des messages reçus reste cohérente avec le flux au fur et à mesure des déplacements du jeton de communication. La seconde primitive kernet_release_io permet de relâcher le jeton. Ce cas correspond à la terminaison d'un des processus partageant une interface de communication sans qu'aucun autre processus n'ait fait de demande d'utilisation de cette interface.

Listing 8.1 – Gestion du jeton de communication

```
void kernet_req_io(struct kernet_socket* socket);
void kernet_release_io(struct °kernetsocket* socket);
```

Nous pouvons donc compléter la définition de la carte des extrémités d'un flux en intégrant la notion de jeton de communication.

Définition 41 ***Carte des extrémités d'un flux:*** *La* ***carte d'extrémités*** *pour une socket d'E/S est la liste des localisations des autres sockets d'E/S, détenant un jeton de communication du flux.*

Chaque extrémité du flux peut disposer de plusieurs *sockets d'E/S* mais ne dispose que d'un seul jeton. Ainsi, par abus de language, la localisation d'une extrémité est assimilée à celle de sa socket d'E/S détenant le jeton de communication.

La carte des extrémités d'un flux permet à la *socket d'E/S* détenant le jeton d'une extrémité distribuée de localiser précisément son correspondant. Chaque *socket d'E/S*

détenant un jeton dispose d'une carte des extrémités à jour. Ainsi, si nous considérons deux processus communiquant entre eux et s'exécutant sur deux nœuds **N1** et **N2**(voir figure 8.2), chacun des ces processus est lié à une extrémité du flux. Chaque *socket d'E/S* dispose d'une carte permettant de localiser l'autre extrémité du flux. Par ailleurs, l'extrémité 2 dispose de plusieurs *socket d'E/S* (nœud **N2** et nœud **N4**) : le jeton de communication permet de localiser celle réalisant les opérations d'envoie ou de réception de messages.

Par conséquent, le déplacement d'un jeton doit maintenir à jour l'ensemble des cartes existant au sein du flux dynamique. Grâce à ces cartes, les communications sont maintenues directes malgré les déplacements des jetons. Le protocole de mise à jour des cartes est présenté dans le paragraphe 8.2.4. Ce protocole comprend notamment les opérations assurant qu'aucun message n'est perdu lors du déplacement de jeton.

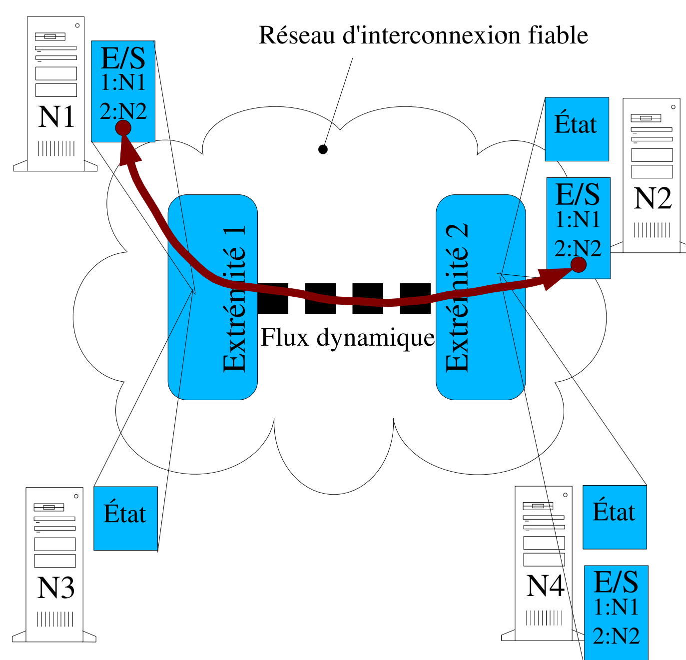


Fig. 8.2 – Flux dynamique

Les informations contenues dans les cartes sont pour chaque extrémité du flux : l'identifiant d'un nœud ou un indicateur de valeur non définie. Ce dernier cas signifie simplement que le jeton n'est pour le moment pas attribué. Par conséquent, les messages ne peuvent être acheminés vers cette destination : le flux est momentanément interrompu. Cette opération est transparente pour l'application cliente qui est simplement suspendue lors de son opération d'émission de message (comme elle pourrait l'être sur un système standard pour d'autres raisons).

De manière simplifiée, l'émission d'une donnée vers une extrémité (voir figure 8.3) se résume à (le cas échéant) attendre qu'un identifiant de nœud soit renseigné dans la carte et à émettre vers cette destination. Ainsi en cas de déplacement d'un processus

A communiquant avec un processus B, ce dernier est mis en attente de la nouvelle localisation avant de poursuivre ses échanges de données.

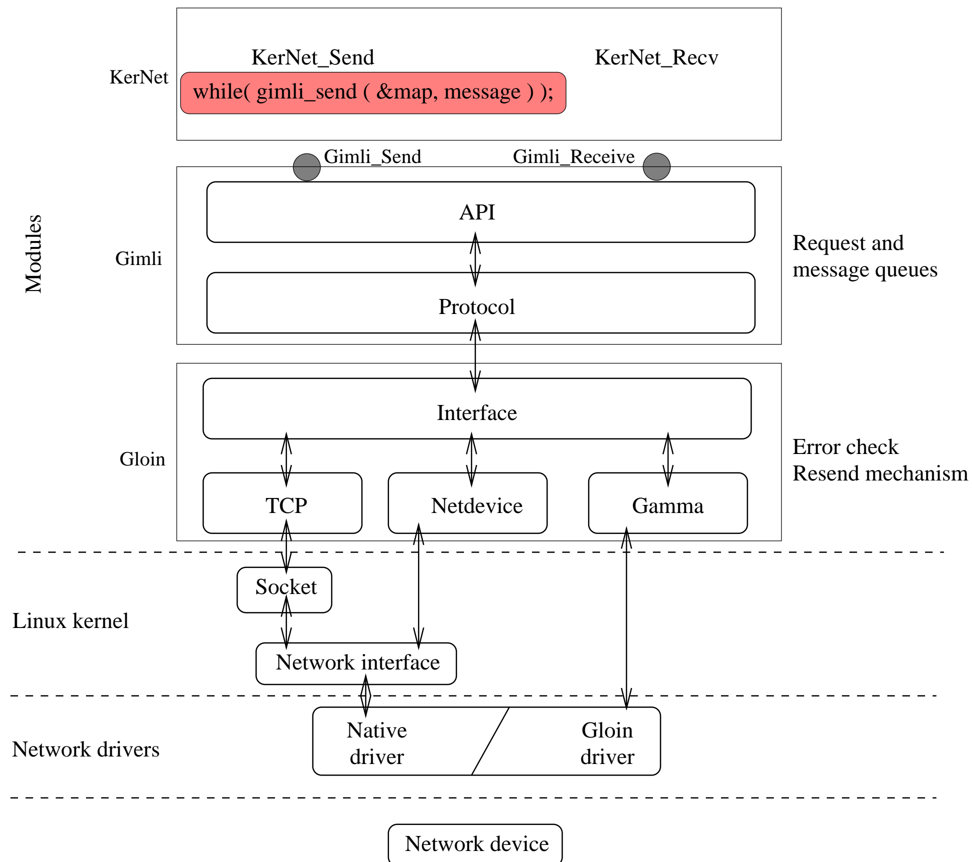


FIG. 8.3 – Communication avec un flux dynamique

8.2.4 Localisation des extrémités d'un flux

Lors du déplacement d'un processus associé à une extrémité par une socket d'E/S, de nombreuses contraintes doivent être respectées :

- le jeton de communication doit être déplacé,
- les cartes d'extrémités existant dans le flux doivent être mises à jour,
- les messages en cours de transfert ne doivent pas être perdus.

Lors d'un transfert de message entre deux extrémités, seule la localisation des jetons de communication de chaque extrémité importe. Par conséquent, le déplacement d'un processus attaché à des sockets mais sans jeton de communication, n'influe pas sur le fonctionnement du flux.

La localisation des jetons de communication des extrémités d'un flux est disponible dans la carte des extrémités. Une carte des extrémités contient, pour chaque extrémité existant dans le flux, l'identifiant et le numéro du nœud de la socket d'E/S détenant

le jeton. Si pour une extrémité, le jeton associé n'est attribué à aucune socket d'E/S, alors une valeur spéciale est fournie à la place du numéro de nœud.

Lors d'une transmission de message, le destinataire correspond à la localisation de l'extrémité présente dans la carte. S'il s'agit du code spécial, la communication est suspendue jusqu'à mise à jour de la carte avec une véritable localisation. Ainsi, à tout moment, la mise à jour de la carte peut interrompre ou re-démarrer les échanges de messages au sein d'un flux.

Le dernier point à résoudre concerne les messages en cours de transmission. Un message en cours de transmission, est un message qui a été confié à l'infrastructure de protocole ou au matériel de communication lui-même. Par conception, une trame se situe obligatoirement dans l'une des trois situations suivantes :

- la trame figure dans la file d'émission : le destinataire n'a pas encore acquitté la réception de la trame, elle ne peut donc être retiré de cette liste,
- la trame figure sur le lien physique de communication : par définition le destinataire ne l'a pas encore reçu et donc ne l'a pas encore acquitté. Une copie de cette trame se trouve toujours dans la file d'émission,
- la trame figure dans la file de réception : un acquitement peut être envoyé et la trame peut éventuellement être retirée de la file d'émission. Dans tous les cas, un numéro de séquence permet d'identifier les ré-émissions inutiles.

Ainsi, lors du déplacement d'un jeton de communication, il importe simplement de s'assurer que la file d'émission de l'autre extrémité du flux est vide. Ceci peut être fait en demandant un signal (puis en attendant) à cette autre extrémité lorsque sa file d'émission est vide. Concernant la file de réception, celle-ci détient des messages ou paquets de message non encore délivrés à l'application elle-même. Par conséquent, cette file doit être déplacée avec le jeton de communication.

Schématiquement le déplacement d'un processus se fait en cinq étapes :

1. libération du jeton : aucun nouveau message ne peut être émis sur le flux
2. détachement de la socket,
3. déplacement du processus (à l'aide du mécanisme traditionnel de migration de processus),
4. rattachement de la socket d'E/S et mise à jour de sa localisation auprès du gestionnaire de flux,
5. demande du jeton de communication et mise à jour de l'ensemble des cartes d'extrémités.

Du point de vue du service de communication, le déplacement de l'interface de communication a lieu à la cinquième étape, lors du déplacement du jeton de communication. Le protocol visant à s'assurer qu'aucun message n'est perdu est réaliser pendant cette étape.

Soit une grappe de calculateurs de quatre nœuds. Afin de clarifier notre propos, nous étudions (voir figure 8.4) un flux présent entre deux nœuds (nœud 1 et 2). Le gestionnaire associé au flux se trouve sur un nœud appelé *gestionnaire*. Le processus exécuté sur le nœud 1, est associé à l'extrémité 0 (master = 0) par la socket d'identifiant

3036. De même, le processus sur le nœud 2 est associé à l'extrémité 1 par la socket 3037. La figure 8.4 présente une version optimisée de la description schématique du déplacement du processus entre le nœud 2 et le nœud 3. En effet, la libération du jeton n'est pas indispensable, le mécanisme attribue le jeton à quiconque en fait la demande. Ainsi, le protocole peut directement débuter par le détachement de la socket.

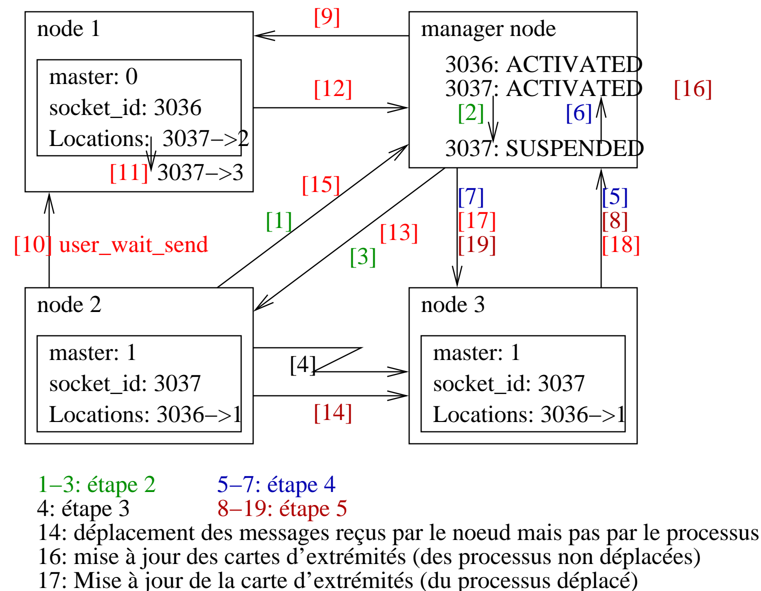


FIG. 8.4 – Protocole de mise à jour des cartes d'extrémités

8.2.5 Interaction avec les modèles de communication

Afin d'exploiter les flux dynamiques, les interfaces de communication doivent être adaptées à cette nouvelle architecture. C'est le rôle des interfaces de protocole que de faire le lien entre les interfaces de communication classiques et l'abstration de flux dynamique.

Gestion des flux dynamiques Dans ce paragraphe, nous présentons les éléments de gestion (voir listing 8.2) d'un flux dynamique.

Listing 8.2 – Primitives de manipulation de flux

```
/* Classe de flux */
struct service_t* kernet_create( struct service_t *sstream, enum kernet_type mode,
                                 enum palantir_interface_id interface_id , int size);
int kernet_destroy(struct service_t* service);

/* Gestion de flux */
struct kernet_stream* kernet_stream_create(struct service_t *service);
int kernet_stream_destroy(struct kernet_stream *stream);

/* Manipulation de socket d'un flux */
```

```
struct kernet_socket* kernet_stream_activate( struct kernet_stream *stream,
                                              struct kernet_duplicate_id *dupid);
int kernet_stream_deactivate(struct kernet_socket *socket);

/* Synchronisation sur un flux */
int kernet_stream_waitok(struct kernet_stream *stream);
```

Afin de manipuler un flux, il est nécessaire de définir l'interface de protocole associée à ce flux. Pour cela, nous définissons une **classe de flux** qui représente l'ensemble des flux associés à un protocole. La création d'une nouvelle classe de flux se fait par kernet_create. Cette fonction permet de caractériser la classe de flux. En plus de définir le protocole associé à la classe, il est possible de définir un mode d'acheminement des données :

- KERNET_DIRECT : le flux ne peut disposer que de deux extrémités et les paquets sont acheminés dans l'ordre de leur émission,
- KERNET_FIFO : le flux peut avoir autant d'extrémités qu'il le souhaite, et les paquets sont acheminés dans l'ordre de leur émission.

Une fois la classe de flux définie, un flux est créé par un appel à kernet_stream_create. Le flux ainsi créé hérite des propriétés déterminées dans sa classe de flux.

La fonction kernet_stream_activate permet l'ajout d'une socket au sein d'une extrémité (déterminée par la parametre dupid) du flux. Si aucune extrémité n'est déterminée, alors une nouvelle extrémité est créée (dans la limite du nombre spécifier à la création du flux). Enfin, la primitive kernet_stream_waitok permet au processus appelant d'interrompre son exécution jusqu'à ce que le flux ait toutes ses extrémités définies. La caractérisation de la socket (socket d'E/S ou socket d'interface) n'est réalisé qu'au moment de l'utilisation de la socket.

Échange de données au sein d'un flux dynamique Dans les interfaces de communication standard, nous pouvons distinguer deux grandes familles. Ces deux familles se distinguent par leur façon de gérer les messages à envoyer. La première famille impose un passage obligatoire par le système d'exploitation pour tout envoi ou réception de message. Schématiquement, les messages sont recopiés dans l'espace d'adressage du système d'exploitation avant d'être remis au périphérique de communication. La seconde famille s'appuie sur un support matériel du périphérique de communication. Seul l'établissement des communications nécessite l'assistance du système d'exploitation. Une fois la connexion établie, les requêtes de communication sont directement transmises au périphérique de communication sans assistance du système d'exploitation. L'avantage de cette dernière approche est un gain dans les latences de communication.

Dans la suite de cette partie, nous étudions comment interagissent les flux dynamiques et ces deux types de systèmes de communication.

8.2.5.1 Communication par appel système

Lorsque toutes les opérations de communication sont réalisées au travers d'appels système spécifiques, ceux-ci peuvent aisément être détournés pour prendre en compte l'architecture des flux dynamiques. Schématiquement, le modèle de communication

consiste à fournir au système d'exploitation des zones mémoire à transmettre à une certaine destination. La destination n'est en général pas manipulée directement, mais au travers d'un descripteur opaque. Le plus souvent ce descripteur est créé à l'établissement du flux de communication. Les zones mémoire sont quant à elles recopiées dans l'espace d'adressage du système d'exploitation avant d'être effectivement transmises.

Dans ce contexte, l'interface des flux dynamiques doit fournir une interface simple pour l'échange de messages. Les fonctions kernet_send et kernet_recv (*listing* 8.3) permettent des échanges de message directs entre extrémités distribuées. Ces fonctions permettent des échanges de messages avec une recopie à l'émission comme à la réception pour le passage de l'espace noyau à l'espace utilisateur. Elles s'assurent que le jeton de communication est bien disponible avant toute communication.

Listing 8.3 – Primitives d'échange de message

```
int kernet_send( struct kernet_socket *socket,
                 void *msg, int msg_len );
int kernet_recv( struct kernet_socket *socket,
                 int flags,
                 void *msg, int msg_len );
```

La réalisation d'une interface de protocole gérée entièrement par appel système consiste donc à confier les requêtes de communication à deux fonctions spécialisées et à gérer la correspondance entre adresse du protocole et extrémité distribuée.

En cas de déplacement du processus communicant, aucune opération particulière n'est à effectuer au niveau de l'interface de protocole si ce n'est de relâcher le jeton de communication. Le reste du protocole de déplacement de communication est pris en charge par les flux dynamiques. Le déplacement de l'interface est considéré comme terminé lorsque celui-ci ré-acquiert le jeton.

8.2.5.2 Communication par accès direct au matériel

À l'instar de la bibliothèque de communication GM de Myricom, les communications par accès direct au matériel se font en deux étapes. Ce type de communication nécessite un support matériel spécifique de la part du périphérique. Dans une première étape, le processus prépare sa communication en réservant des ressources auprès du périphérique de communication et déclare des zones mémoire en tant que zones d'échange de données. Lors de la déclaration, le périphérique enregistre les adresses physiques des zones mémoire concernées de manière à y accéder ultérieurement. Afin de conserver les adresses physiques, ces zones mémoires sont verrouillées dans le SE.

Lors du déplacement du processus communicant, le problème n'est plus uniquement de localiser efficacement la nouvelle destination, mais aussi de remettre en état l'interface de communication sans aucune aide de l'application. Comme il paraît difficilement envisageable de disposer exactement des mêmes zones de mémoire physique sur le nœud de destination et sur le nœud d'origine, un mécanisme de ré-enregistrement de la mémoire d'échange doit être prévu. Dans un SSI, cette opération revient au service de gestion globale de la mémoire. Après le déplacement du processus, de nouvelles zones de mémoire physique doivent être associées aux différentes zones de mémoire virtuelle

du processus. Toutefois, à la différence des zones de mémoire standard du processus, il est nécessaire de re-déclarer les plages de mémoire servant de zone de communication au périphérique.

La migration d'un processus communicant consiste donc à :
- déplacer le contenu des zones d'échange,
- supprimer la déclaration de la zone d'échange du périphérique situé sur le nœud initial,
- ré-instancier le contenu de la mémoire et en particulier de la zone d'échange,
- déclarer les nouvelles adresses physiques de la zone d'échange auprès du nouveau périphérique,
- mettre à jour les cartes des extrémités.

Dans ce descriptif, le déplacement de la zone d'échange est effectué par le mécanisme de migration de processus (sans modification de celui-ci). La problématique de localisation des extrémités mobiles est résolue à l'aide des mécanismes de localisation fournis par les flux dynamiques.

Seuls les mécanismes de suppression de la déclaration (côté émetteur) et d'enregistrement de la zone d'échange (côté récepteur) en fin de déplacement de processus doivent être ajoutés à l'architecture existante.

Nous voyons ici une utilisation conjointe des mécanismes de gestion globale des ressources du SSI permettant une opération originale telle que le déplacement de processus communiquant par accès direct au matériel sans support spécifique de l'application ni du matériel. Le seul changement réalisé est dans une indirection du système d'adressage de la bibliothèque de communication, afin de supporter le mécanime d'adressage des flux dynamiques. Cette modification mineure dans la bibliothèque de communication n'a pas de répercussion sur l'application elle-même. Celle-ci reste globalement inchangée (à l'édition de lien près).

8.2.6 Tolérance aux fautes

Une des approches classiques dans le domaine de la tolérance aux fautes est la sauvegarde/restauration de points de reprise. Comme nous l'avons rappelé dans le chapitre 2, la réalisation d'un point de reprise d'une application quelconque impose la capture de l'état de cette application. L'état d'une application parallèle, fondée sur le modèle de communication par message, se compose de :
- l'état de ses processus,
- l'état de ses canaux de communication (synchronisation des extrémités du canal),
- le contenu de ses canaux de communication.

La capture de l'état d'un processus disposant d'interface de communication est la même que pour un processus quelconque. Le système à image unique Kerrighed dispose d'un support aux points de reprise pour des applications parallèles communicantes par mémoire partagée[12]. Un tel mécanisme réalise une synchronisation globale des processus de l'application (à l'aide d'un coordinateur) puis effectue, processus par processus, un point de reprise de l'application parallèle. L'association d'un tel mécanisme de capture coordonnée d'état des processus d'une application parallèle à un mécanisme de

capture des états des canaux de communication permet la réalisation de points de reprise d'application parallèle communiquant par échange de messages.

Dans un système tel que Kerrighed, la difficulté réside donc dans la capture des états liés aux canaux de communication. Grâce aux flux dynamiques, il est aisé d'extraire les caractéristiques d'un flux et en particulier la liste des extrémités et leur localisation. Ces informations doivent être ajoutées à celles réunies lors du calcul du point de reprise coordonné des processus de l'application parallèle.

Le dernier problème réside dans la capture du contenu des canaux de communication. Le contenu d'un canal de communication est distribué en plusieurs endroits : la file des messages en cours d'émission, les paquets de données présents sur le lien physique de communication et les messages en cours de réception. Puisque le modèle de reprise ne permet pas de re-démarrer qu'un seul processus de l'application, il n'est pas nécessaire de conserver un historique de chaque canal de communication. En fait, il suffit de s'assurer qu'aucun message n'est en cours de transmission sur le réseau physique lors de la synchronisation des processus en vue du calcul de point de reprise coordonné. Ainsi, après l'étape de synchronisation, les liens physiques de communication sont vides. Seuls les files d'émission et de réception des messages sont à ajouter au point de reprise. Lors d'une reprise de l'application, ces files d'émission et de réception sont restaurés. Il n'y a pas de risque de perte de message car tout message se trouve au minimum dans l'une de ces deux files (selon son acquittement). De même, il ne peut y avoir de duplication de message, car les numéros de séquence permettent d'identifier les paquets déjà acquittés (et donc reçus) des paquets non acquittés (et non encore reçus).

Notons que ces opérations de calcul d'état des canaux de communication ne nécessitent aucun support de la part des applications. Par conséquent, il est possible de calculer des points de reprise en toute transparence vis-à-vis des applications.

8.2.7 Intégration dans l'architecture de communication pour SSI

Par rapport à l'architecture globale des communications du système à image unique, les communications pour les applications sont gérées au sein d'un protocole de communication dédié. Ce protocole se place au même niveau que celui destiné aux communications pour le SE (présenté dans le chapitre 7) dans l'infrastructure pour les protocoles de communication.

8.3 Communication extra-grappe de calculateurs

Dans le paragraphe précédant, nous avons proposé un mécanisme pour un support efficace des communications au sein d'une grappe de calculateurs. Toutefois, une grappe de calculateurs ne réalise pas exclusivement des communications internes. En particulier, lorsqu'un service réseau est déployé sur la grappe, des communications entre un nœud interne à la grappe et une machine quelconque externe à la grappe ont lieu. Une problématique analogue existe dans le domaine de la continuïté de service. En effet, lorsqu'un même service est simultanément disponible sur plusieurs machines serveur,

l'objectif de continuïté de service amène, en cas de défaillance d'un serveur, à vouloir déplacer les communications en cours, vers un serveur correct.

Le travail présenté dans cette partie a été réalisé dans le cadre de l'activité de recherche sur les **backdoors**[88, 89, 90] menée par Florin Sultan sous la direction de Liviu Iftode, pendant le séjour que j'ai effectué au cours de l'été 2003 dans le laboratoire Discolab de Rutgers University.

Les *backdoors* constituent une approche originale au problème des systèmes auto-réparants. Les systèmes auto-réparants se caractérisent notamment par une capacité à s'auto-surveiller, à diagnostiquer des fautes et à pouvoir revenir dans un état correct sans intervention humaine. L'architecture des *backdoors* est fondée sur des mécanismes d'accès distant à la mémoire (en lecture comme en écriture) afin de réaliser un système non-intrusif (par rapport aux processeurs surveillés) de surveillance et de réparation des services rendus par la machine. Les applications visées sont celles qui fournissent un service par l'intermédiaire du réseau de communication (serveurs web ou serveurs de base de données). Un tel système a pour ambition de réparer des fautes matérielles (processeurs, réseau, etc.) ou logicielles (système d'exploitation, applications, denis de service, etc.).

8.3.1 Infrastructure pour l'accès à distance aux structures de données d'un SE

L'accès distant à la mémoire d'une machine constitue l'hypothèse centrale des *backdoors*. Afin de pouvoir détecter et réagir dans toutes les situations, la contribution des processeurs principaux de la machine surveillée doit être minimale. Il est donc exclu d'employer un processus réalisant localement le travail désiré par une machine distante.

Technologie R-DMA La technologie R-DMA (*Remote-DMA*) permet de réaliser un accès matériel à la mémoire d'une machine. Sur un bus de communication tel que le bus PCI, les échanges de données se font généralement à l'initiative du processeur central. Cependant, rien dans la spécification PCI n'interdit à une carte fille PCI de déclencher un accès mémoire de sa propre initiative. L'amélioration des technologies en micro-électronique a permis d'embarquer sur une carte PCI fille, un processeur autonome, une mémoire dédiée à ce processeur ainsi qu'un contrôleur DMA pour les accès à la mémoire centrale. Ces caractéristiques ont été initialement mises en place afin de décharger le processeur central d'une part des tâches de communication. Une lecture distante correspond au fait de demander au processeur embarqué d'accéder à une zone mémoire puis de la transférer par le réseau.

Lecture/Écriture de structures de données d'un SE Les accès de type R-DMA se font directement de mémoire physique à mémoire physique, le processeur embarqué n'ayant pas de notion de la mémoire virtuelle du SE. Il est donc possible d'accéder à l'ensemble de la mémoire physique disponible sur la machine visée. Dans le cas de structures de données internes au système d'exploitation Linux, la mémoire virtuelle utilisée est statique dans le temps : les adresses physiques correspondantes ne varient

donc pas. En partant de ce constat, il est possible de lire le contenu de la mémoire d'un système d'exploitation distant simplement en 'suivant' les références mémoire. Suivre une liste de descripteurs de processus au travers d'un réseau devient aussi simple que de lire celle disponible en local. Les opérations d'écriture à distance se font de manière analogue aux opérations de lecture.

Prise de verrou à distance Dans la plupart des systèmes d'exploitation, il existe un mécanisme de synchronisation fondé sur l'attente active du processeur. Un tel mécanisme repose sur la lecture continue d'une valeur mémoire jusqu'au changement de cette valeur. Par conséquent, moyennant quelques précautions que nous décrivons dans la suite de ce paragraphe, il est possible de "prendre" un verrou à distance.

8.3.2 Application aux déplacements de sockets TCP/IP

Dans un contexte de défaillance d'une machine hébergeant un service communiquant par TCP/IP, la solution retenue consiste à déplacer l'adresse IP complète du nœud défaillant vers un nouveau nœud.

Ce déplacement se fait par simple lecture des structures de données internes du SE associé aux sockets. Une fois l'état capturé, ces informations sont ré-instanciées sur la machine de destination.

8.4 Communication inter-grappes de calculateurs

Nous avons précédemment décrit deux mécanismes permettant la migration de flux. Ces deux mécanismes sont développés dans deux approches différentes du problème de déplacement de flux. Dans le premier cas, les interfaces de communication sont respectées au détriment de leur protocole. Ainsi, tous les types d'interface peuvent être déplacés (y compris les interfaces traditionnellement locales à un nœud) mais en contre-partie, une interface mobile ne peut communiquer avec la même interface fournie en standard par un système d'exploitation classique. Dans la seconde approche, le protocole de communication est respecté, mais l'approche employée est spécifique au protocole TCP/IP.

Les communications entre deux grappes de calculateurs peuvent être ramenées à l'un des deux cas précédemment étudiés. En effet, soit les communications sont réalisées entre deux systèmes de communication partageant un mécanisme d'adressage commun des nœuds (par exemple adresse IP, avec une primitive de communication TCP). Dans ce cas, un mécanisme tel que les flux dynamiques peut permettre le déplacement de chaque extrémité du flux de communication au sein de sa propre grappe de calculateurs. Soit, il n'y a pas d'espace de nommage commun aux nœuds des deux grappes de calculateurs, et un mécanisme spécifique au protocole de communication souhaité doit être déployé (comme dans le cas des communications extra-grappes).

8.5 Résumé

Nous avons présenté dans ce chapitre une architecture de communication offrant un support adapté aux différents types de communication possibles au sein et en relation avec une grappe de calculateurs exécutant un système d'exploitation distribué de type système à image unique. L'objectif de cette architecture est de permettre le déploiement, l'exécution, le déplacement et la réalisation de points de reprise d'applications parallèles fondées sur le modèle de communication par message, et ceci en toute transparence vis-à-vis de ces applications.

La notion de *flux dynamique* est une solution à ce problème. Les flux dynamiques permettent de fournir une base de communication efficace car ils effectuent toujours des communications directes entre l'émetteur et le récepteur. L'abstraction des flux dynamiques, orientée exclusivement vers la localisation des extrémités du flux, permet la réalisation de l'ensemble des interfaces de communication standard à un système d'exploitation, ainsi que l'extension d'interfaces plus spécifiques telles que le système de communication GM de Myricom. Les flux dynamiques s'appuient sur le concept d'*extrémités distribuées* permettant une séparation des opérations de changement d'état du flux des opérations de communication (envoi/réception de message). Par ailleurs, un mécanisme d'exclusion mutuelle permet d'identifier à tout moment le destinataire d'un message. En cas de changement de destinataire, les messages émis mais non encore délivrés à l'application sont transmis au nouveau destinataire en même temps que le jeton d'exclusion mutuelle.

Par ailleurs, le choix de placer les flux au sein des interfaces de communication, sans obligatoirement étendre celles-ci, permet une véritable transparence de nos mécanismes pour les applications visées. Cette transparence est à la fois au niveau des sources de l'application, des binaires générés et des bibliothèques associées à celle-ci.

Enfin, en s'inscrivant dans le cadre plus général des systèmes à image unique, notre architecture permet, en coopération avec le système de gestion globale de la mémoire, le déplacement de processus exploitant des périphériques de communication sans passer par le système d'exploitation.

Chapitre 9

Éléments de mise en œuvre

Nous avons présenté, dans les chapitres précédents, les différentes abstractions liées à l'architecture de communication que nous proposons pour un système à image unique pour grappe. Tout d'abord, nous avons présenté les modèles de matériel et de protocole de communication (chapitre 6) définissant l'ensemble des hypothèses sur lesquelles sont contruites nos solutions. Puis, nous avons présenté deux contributions. La première est relative aux communications de système d'exploitation à système d'exploitation, les transactions de communication (chapitre 7). La seconde concerne les processus (mobiles) communiquant par message (chapitre 8).

Nous avons mis en œuvre le système de communication que nous avons conçu dans un prototype lié au système à image unique Kerrighed. L'intégration complète dans la version officielle de Kerrighed a été réalisée avec le concours de David Margery. David a, par ailleurs, réalisé le support de l'interface de communication des pipe. Nous donnons dans ce chapitre quelques éléments de mise en œuvre. Dans un premier temps, nous décrivons l'interface de programmation d'un nouveau protocol au sein de notre architecture de communication. Nous explicitons, dans un second temps, différents types d'interfaces construites à partir des transactions de communication. Puis, nous détaillons la mise en œuvre des interfaces de communication à l'aide des flux dynamiques.

9.1 Système de communication de niveau protocole

La figure 9.1 présente une synthèse des primitives de programmation à fournir afin de concevoir un protocole de communication. Ces primitives ont été introduites et décrites dans le chapitre 6. Le modèle de matériel est fourni par les fonctions de gloin. Gimli/Ack est le protocole assurant la gestion des acquittements des trames dans le système de communication. Gimli/Kernel et Gimli/User sont respectivement les protocoles dédiés aux communications pour les services système distribués, et au support aux applications communiquant par flux.

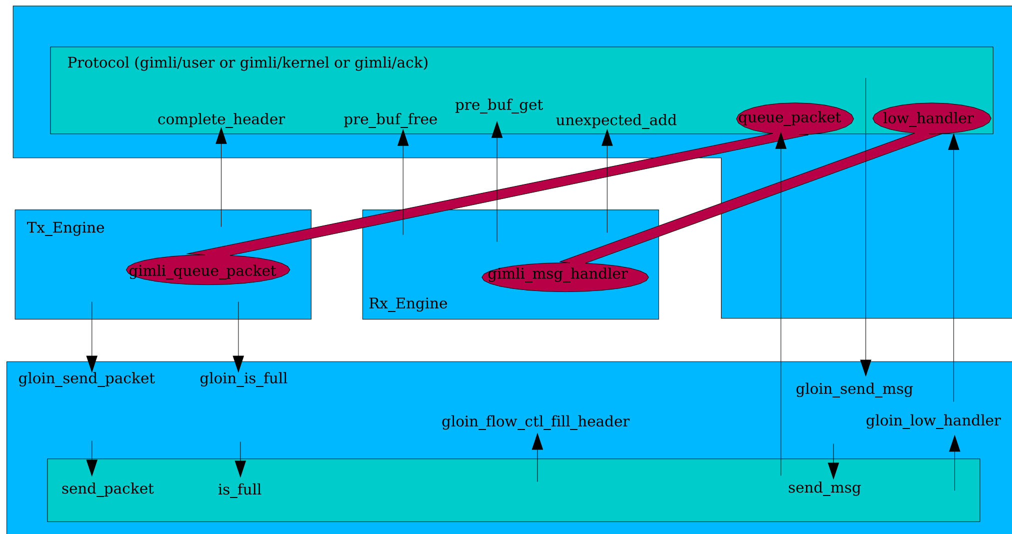


Fig. 9.1 – Interface de programmation pour l'infrastructure de protocole

9.2 Interfaces de communication pour les services système distribués

Les transactions de communication permettent de décrire le comportement d'un service par rapport à ses requêtes de communication. Dans ce paragraphe, nousr présentons dans un premier temps un exemple complet de service système distribué exploitant les transactions de communication. Dans un second temps, nous présentons un ensemble d'interfaces simplifiées conçues à partir des transactions de communication.

9.2.1 Exemple de mise en œuvre d'un service système distribué à l'aide des transactions de communication

Considérons la réalisation d'un service de type *ping-pong* mesurant la duré liée à plusieurs échanges d'un message entre deux nœuds au sein d'une grappe de calculateurs.

La mise en œuvre classique de ce modèle consiste à exploiter deux processus de niveau utilisateur : un pour le client et l'autre pour le serveur. Dans ce modèle, les processus pilotent la progression de l'échange de messages (voir figure 9.2). Ainsi, à chaque réception de message, le système de communication active le processus destinataire correspondant afin que celui-ci décide d'un éventuel renvoi du message.

Dans le service que nous venons de décrire, seul le temps mesuré importe au processus. Par conséquent, dans un tel service, le système de communication, pourrait faire progresser la circulation du message, de manière autonome vis-à-vis des processus (voir la figure 9.3).

Le mécanisme de transaction de communication permet au concepteur du service de décrire les opérations qui vont suivre le premier envoi de message ou la première réception de message. Pour chacune des réceptions réalisées dans ce ping-pong, une

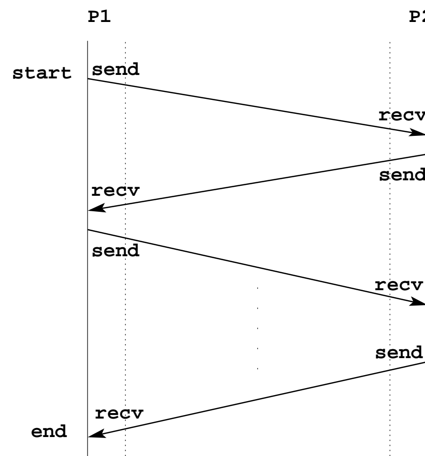


FIG. 9.2 – Réalisation classique de ping-pong

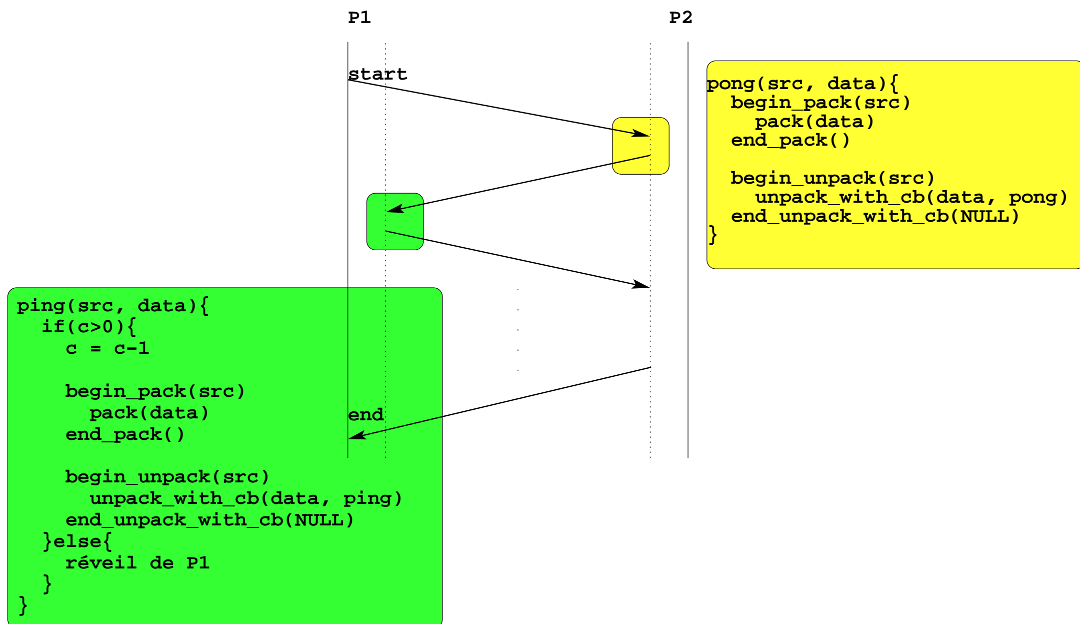


FIG. 9.3 – Exemple de ping-pong à l'aide des transactions de communication

action est postée afin de déterminer la suite du traitement du service. Cette action est exécutée lorsqu'un message est réceptionné par le système de communication. Du côté serveur (P2), le traitement consiste simplement à ré-emettre le message et à préparer la réception suivante en indiquant la prochaine action souhaitée (fonction *pong*). Du côté client (P1), la ré-émission est conditionnée par le nombre d'aller/retour voulu dans le service. Lorsque ce nombre est atteint, le processus client est réveillé par le système de communication. Dans le cas où il reste des échanges à réaliser, le système de communication renvoie le message et poste la réception suivante en y associant le traitement voulu (fonction *ping*).

Dans notre exemple, le modèle fourni par les transactions de communication permet de ne jamais solliciter les processus **P1** et **P2** pendant l'exécution du service.

9.2.2 Interfaces complémentaires

Les transactions de communication représentent le modèle le plus complet que nous ayons conçu pour les services système distribués. Les transactions de communication se trouvent au cœur de tous les échanges de messages au sein des services système distribués. L'envoi ou la réception de messages par les transactions de communication nécessitent une préparation du système de communication. Lors de l'envoi de messages non composés, l'interface des transactions de communication peut s'avérer lourde à utiliser. Des versions simplifiées de cette interface ont été réalisées afin d'alléger l'usage des transactions de communication lorsque l'aspect "composition de message" ou "composition d'action" n'est pas requis. Dans la suite de ce paragraphe, nous présentons ces différentes interfaces.

send/receive non bloquant Il s'agit du modèle de communication asynchrone décrit dans le paragraphe 3.2. Les demandes d'émission ou de réception de message sont transmises au système de communication. La synchronisation avec le résultat de l'opération est réalisée ultérieurement. Ces primitives évitent une gestion directe du descripteur de communication. Le *listing* 9.1 décrit en pseudo-langage les opérations liées aux transactions de communication.

Listing 9.1 – Primitives asynchrones

```
void* gimili_async_send_lt(
                kerrighed_node_t src, gimli_channel_t channel,
                gimli_schannel_t schannel,
                void * buffer, size_t length){
  desc = gimli_begin_pack(src, channel, schannel);
  gimli_pack(desc, 0, buffer, length);
  return desc;
};

ssize_t gimli_wait_send(void* handler){
  gimli_end_pack(desc);
};

void* gimli_async_recv_lt(
                kerrighed_node_t src, gimli_channel_t channel,
                gimli_schannel_t schannel,
```

```
                  void * buffer, size_t length,
                  com_status_t * status){
  desc = gimli_begin_unpack(src, channel, schannel);
  gimli_unpack(desc, 0, buffer, length);
};

ssize_t gimli_wait_recv(void* handler){
  gimli_end_unpack(desc);
};
```

Les actions réalisées par ces fonctions sont très simples. Il s'agit simplement d'opérations liées à un changement d'interface.

send&forget avec réception bloquante Le modèle de programmation que nous venons d'introduire, impose de manipuler les zones tampons avec précaution. En effet, les tampons ne doivent pas être modifiés tant que la primitive de synchronisation gimli_wait_* n'a pas été appelée. Toutefois, afin de simplifier l'usage de l'interface de communication, il est souhaitable d'employer des primitives permettant d'exploiter les tampons dès le retour du système de communication. Les fonctions présentées dans la figure 9.2 réalisent les opérations nécessaires (poster la requête de communication et attendre l'acquittement de l'opération) pour un usage sans contrainte sur le tampon de communication.

Listing 9.2 – Interface de communication send/receive

```
gimli_send(kerrighed_node_t src, gimli_channel_t channel,
           gimli_schannel_t schannel,
           void * buffer, size_t length){
  desc = gimli_begin_pack(src, channel, schannel);
  gimli_pack(desc, GIMLI_TRANSACT_SAFE, buffer, length);
  gimli_end_pack(desc);
};

gimli_recv(kerrighed_node_t src, gimli_channel_t channel,
           gimli_schannel_t schannel,
           void * buffer, size_t length,
           com_status_t * status){
  desc = gimli_begin_unpack(src, channel, schannel);
  gimli_unpack(desc, GIMLI_TRANSACT_WAIT, buffer, length);
  gimli_end_unpack(desc);
};
```

Ce modèle de programmation fut le premier modèle disponible dans Kerrighed. De nombreux services distribués dont le service de gestion globale de la mémoire et celui de la gestion globale des processus emploient ce modèle. Par conséquent, il est nécessaire de le conserver au moins à titre de compatibilité avec l'existant.

9.3 Système de communication par flux dynamiques

Le chapitre 8 présente les flux dynamiques comme support aux déplacements et à la sauvegarde de points de reprise d'applications communiquant par échange de messages. Cependant, les applications existantes susceptibles d'être exécutées sur un SSI tel que

Kerrighed n'emploient pas de flux dynamique. Les extrémités des flux dynamiques constituent les liens entre le SE Linux et le mécanisme de flux dynamique.

Dans la suite de ce paragraphe, nous décrivons deux interfaces de communication classiques dans un système Unix : les pipe et les sockets Unix. En particulier, nous décrivons comment mettre en œuvre les versions distribuées de ces interfaces en utilisant les flux dynamiques.

9.3.1 Communication par pipe

L'interface de communication par pipe est un système de communication permettant à deux processus localisés sur une même machine de communiquer par échange de messages. Le protocole pour la mise en place d'un pipe en est donc simplifié, et se résume simplement à un appel système : pipe. Les pipes sont, par exemple, fréquemment utilisés pour permettre à un processus père de communiquer avec un processus fils.

L'appel système pipe (voir figure 9.4) crée une paire de descripteurs de fichier. Chacun de ces descripteurs pointe vers une extrémité du pipe : l'une pour lire et l'autre pour écrire. Un processus **P1** crée un pipe et reçoit les deux descripteurs associés à ce pipe. Lors de la création du processus fils **P2** par l'appel système fork, le processus fils hérite des descripteurs de fichier. Le bon usage des pipe recommande à chaque processus de fermer le descripteur non utilisé.

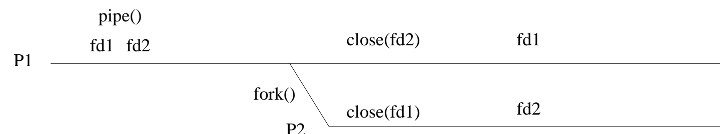


FIG. 9.4 – Usage classique d'un pipe

La réalisation de l'interface pipe à l'aide de flux dynamiques se fait simplement en créant un flux dynamique à deux extrémités. Chaque extrémité est associée à un descripteur de fichier (voir figure 9.5). Les lignes `(1)`, `(2)`, `(3)` et `(4)` correspondent aux appels système réalisés par l'application. Les lignes `(1.1)` à `(1.3)` détaillent les opérations effectuées par le système de communication lors de l'appel pipe. Enfin, `(4.1)` montre le branchement direct entre les opérations de lecture/écriture de l'interface et celles d'émission/réception des flux dynamiques.

9.3.2 Communication par socket unix

Les sockets Unix constituent une autre forme de communication interne à une machine. Alors que les pipe ne permettent la réalisation de communication qu'entre processus disposant d'un lien de filiation, les socket unix permettent à deux processus quelconques de créer un canal de communication. L'adressage entre ces deux processus est réalisé par un support commun : le système de fichier.

Dans le SE Linux, les sockets unix sont simplement réalisées en bénéficiant du partage physique de la mémoire. Les structures internes de ces interfaces étant accesssibles, il

est aisé pour le SE de déplacer une trame de l'émetteur vers le récepteur. Dans un SSI tel que Kerrighed, il n'est pas possible d'accéder directement aux structures de données existant sur un autre nœud. C'est ici qu'interviennent les flux dynamiques et les interfaces de protocole.

La figure 9.6 représente l'automate le plus simple associé à l'utilisation de sockets unix. Rien dans les flux dynamiques ne permet de gérer une liste de pareil état. La gestion du protocole spécifique à une interface de communication donnée est dévolue aux interfaces de protocole.

Lors de la définition d'un nouveau type d'interface de protocole, la classe de flux associée est déclarée auprès du mécanisme de gestion de flux dynamique. Cette étape de déclaration permet la définition des propriétés communes (type de flux, nombre d'extrémités, etc.) associées à l'interface de protocole considérée.

Lorsqu'un processus effectue un appel système de type accept sur une socket, l'interface de protocole `(i)` crée le flux dynamique, `(ii)` associe une extrémité du flux au processus appelant et `(iii)` attend (par kernet_stream_waitok) que l'autre extrémité soit elle aussi, associée à un processus. Supposons qu'un autre processus (éventuellement sur un autre nœud) effectue l'appel système connect sur cette socket Unix. La seconde extrémité est alors associée, et l'interface de protocole peut libérer le premier processus en attente. Après réalisation de ces opérations, le nouveau flux dynamique est disponible et peut transporter directement des messages entre ses extrémités.

Le protocole relatif à la séquence accept/connect est un bon exemple de la séparation entre protocole et transmission de données. Le protocole est réalisé par les interfaces de protocole alors que la transmission de données est à la charge des flux dynamiques. La même approche a été employée pour programmer les autres opérations disponibles sur une socket : poll (afin de permettre l'usage de la primitive de multiplexage des opérations de communication select), listen, etc.

9.4 Résumé

Ce chapitre nous a permis de présenter les éléments nécessaires à l'utilisation de l'architecture de communication que nous proposons pour les systèmes à image unique pour grappe.

Cette architecture est suffisament souple pour :

- permettre la conception de nouveaux protocoles de communication,
- offrir les modèles de programmation habituels à partir des transactions de communication,
- supporter de nouvelles interfaces de communication pour les applications communicantes.

L'architecture logicielle retenue consiste à séparer une fonctionnalité (par exemple le modèle de communication, les transactions de communication ou encore les flux dynamiques) des interfaces logicielles intégrant ces fonctionnalités au sein du système existant. Une telle approche permet d'envisager la ré-utilisation de l'ensemble des concepts présentés dans ce document, dans d'autres SSIs ou plus généralement dans d'autres

systèmes d'exploitation distribués pour grappe de calculateurs.

```
        P1                                   P2
(1)     pipe(fd[2])
(1.1)       |  stream = create(DIRECT, 2);
(1.2)       |  fd[1] = attach(stream);
(1.3)       |  fd[2] = attach(stream);
            +----
(2)     fork()
(3)     close(fd2)                           close(fd1)
(4)     write(fd1,...)                       read(fd2,...)
(4.1)       +  kernet_send(...)                  + kernet_recv(...)
```

FIG. 9.5 – Basic pipe implementation

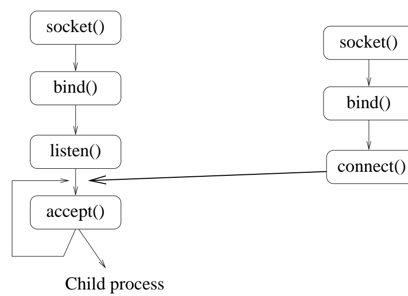


FIG. 9.6 – Protocole associé aux sockets Unix

Chapitre 10

Évaluation du système de communication de Kerrighed

L'architecture de communication proposée a été mise en œuvre dans le cadre du système à image unique Kerrighed. La première partie de cette architecture, liée aux communications pour les services systèmes distribués est complètement réalisée. Toutefois, l'ensemble des services systèmes distribués de Kerrighed exploitent encore l'interface simplifiée, limitant ainsi l'évaluation complète de notre prototype. La première partie de ce chapitre est consacrée à l'évaluation des transactions de communication au sein d'un service simplifié. Par ailleurs, l'intégration des flux dynamiques permet un support efficace à l'exécution d'applications parallèles communiquant par échange de messages. Nous présentons dans la seconde partie de ce chapitre une évaluation du mécanisme de flux dynamique pour différents matériels d'interconnexion ainsi que pour différentes interfaces de communication. La troisième partie du chapitre est consacrée à une comparaison, en terme de performance des approches respectives pour la gestion des flux, dans les systèmes à image unique OpenSSI, openMosix et Kerrighed.

10.1 Évaluation du système de communication pour les services système distribués

Le mécanisme de transaction de communication a été conçu pour permettre la réalisation de services système distribués les plus efficaces et les plus réactifs possible. Dans cette partie, nous comparons deux versions du service de ping-pong décrit dans le paragraphe 9.2.1 : une version classique utilisant des processus ainsi qu'une version exploitant les transactions de communication. L'objectif de cette évaluation est de comparer l'évolution des performances du service distribué en fonction d'une charge de calcul créée sur l'un des nœuds.

La plateforme de test est composée de deux machines (*Pentium III*, 500 MHz) interconnectées par un réseau Gigabit Ethernet. Une charge artificielle est créée sur ces machines en exécutant une boucle infinie dont nous faisons varier le niveau de priorité.

La figure 10.1 présente ces résultats. En abscisse nous plaçons le niveau de priorité

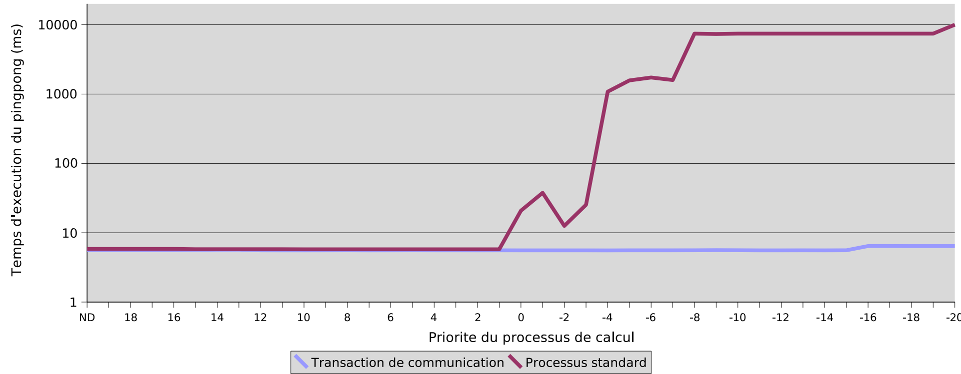


Fig. 10.1 – Evaluation du service de ping-pong réalisé à l'aide des transactions de communication

de la charge (-20 correspond à une charge de priorité maximale et 20 correspond à une charge de priorité minimale). En ordonnée nous présentons le temps d'exécution du service de ping-pong.

Nous constatons que le mécanisme de ping-pong exploitant les transactions de communication est pratiquement insensible au niveau de priorité de la charge. À l'inverse, les performances du ping-pong utilisant des processus classiques se dégradent fortement à mesure qu'augmente le niveau de priorité de la charge. Ainsi dans le cas des transactions de communication, une grande réactivité des services système est obtenue permettant, le cas échéant, de corriger un problème de charge sur un nœud de la grappe qui ne serait plus utilisable par des processus classiques (*i.e.* d'un dénis de service par la charge).

10.2 Évaluation des communications par flux dynamique

L'évaluation du mécanisme de flux dynamique a pour objectif de déterminer le coût de notre architecture pour des applications parallèles communiquant par échange de messages. Le second axe de notre évaluation vise à déterminer le coût d'une migration de processus au sein de l'application parallèle.

Les performances de notre architecture sont déterminées par les performances du système de communication de bas niveau ainsi que par le surcoût lié au mécanisme de flux dynamique. Par conséquent, les performances mesurées sont bornées par celles du système de communication de bas niveau.

Nous avons utilisé comme application de test l'application NetPipe[79]. Cette application réalise un échange de messages de type *pingpong* entre deux processus, en faisant varier la taille des messages. Les processus peuvent être placés sur le même nœud, ou sur deux nœuds distincts. Lors de l'évaluation du système de communication dédié aux applications, nous utilisons l'environnement de communication MPI et en particulier la version MPI de NetPipe.

Lorsque les processus sont placés sur le même nœud, les interfaces d'interconnexion peuvent être :

– l'interface locale Inet (socket TCP/IP – *vanilla*-TCP),
– l'interface conçue pour la mémoire partagée (socket unix – *vanilla*-Unix),
– l'interface de bas niveau KerNet TCP (TCP-like),
– l'interface de bas niveau KerNet Unix (socket unix distribuée).

L'appellation *vanilla* correspond à l'usage d'une version non modifiée du système d'exploitation Linux. Par conséquent, *vanilla*-TCP et *vanilla*-Unix sont les versions non modifiées telles qu'elles sont disponibles dans les versions officielles de Linux.

La grappe de calculateurs utilisée est composée de PC à base de *Pentium III* (500 MHz, 512Ko cache) dotés de 512Mo de mémoire physique. Lorsque les processus sont placés sur des nœuds distincts, les systèmes d'interconnexion retenus pour nos évaluations, sont FastEthernet et Gigabit Ethernet.

Les graphiques suivants montrent les courbes de performance de l'application NetPipe en fonction des diverses interfaces de communication et des différents réseaux d'interconnexion. De plus, nous ajoutons à chacun de ces résultats, la courbe de performance brute de la partie basse du système de communication. Cette courbe sert de référence pour évaluer le surcoût de l'architecture des flux dynamiques par rapport à des communications statiques.

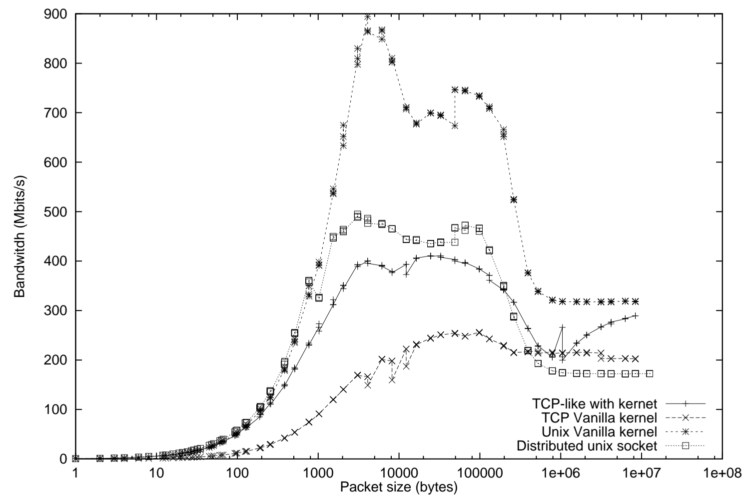


FIG. 10.2 – KerNet : Performance locale à un nœud

Communication interne à un nœud Lorsque deux processus communicants se situent sur le même nœud (figure 10.2), les flux dynamiques sont plus performants que les sockets TCP du système standard. Ceci est principalement dû à la taille réduite de la pile réseau traversée dans KerNet. Pour ce type de communication, certaines fonctionnalités de TCP/IP sont superflues et pénalisent les performances globales de TCP/IP. C'est pour cette raison que les systèmes Unix disposent d'une version de

l'interface socket adaptée pour les communications locales : ce sont les sockets Unix. Comparés à l'interface des sockets Unix, les flux dynamiques n'atteignent pas les mêmes performances. Ceci est principalement lié à notre système de communication qui effectue dans tous les cas, des allocations/désallocations de mémoire.

Communication entre deux nœuds distincts Les figures 10.3 à 10.5 présentent les résultats de l'évalution du système de flux dynamique entre deux nœuds distincts.

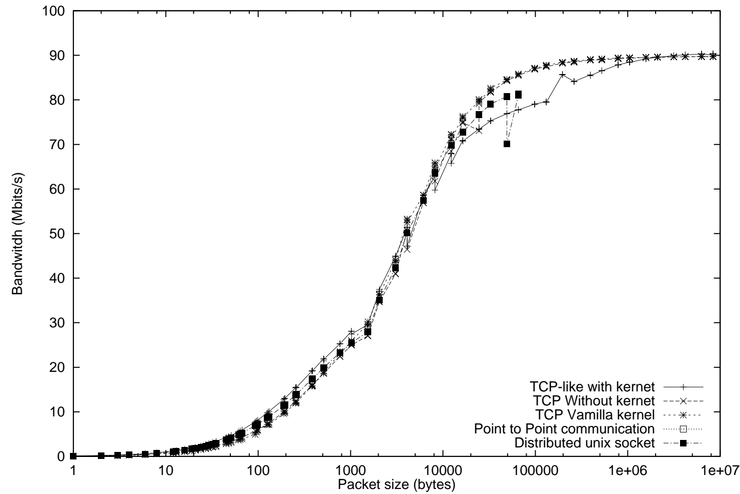


FIG. 10.3 – Performance sur FastEthernet

Nous notons que les interfaces ont un faible impact sur les performances des flux dynamiques : les interfaces de type sockets Unix ou TCP/IP offrent approximativement les mêmes valeurs.

Avec un réseau *FastEthernet*, la bande passante des flux dynamiques est similaire à celle du système de communication de bas niveau (figure 10.3).

Avec la version courante du système de communication de bas niveau, plusieurs recopies sont effectuées lors de la transmission du message. Lors de l'utilisation de réseaux d'interconnexion récents (Gigabit Ethernet, Myrinet), la latence induite par les recopies devient significative par rapport à celle du réseau physique. Les performances mesurées s'en trouvent fortement pénalisées (figure 10.4).

Une évaluation préliminaire a été effectuée avec un réseau Myrinet. Pour ces tests, nous avons utilisé une interface de communication de bas niveau de type Ethernet (appelée *netdevice* au sein du noyau Linux). Cette interface utilise l'interface de compatibilité de Myrinet et non pas son interface GM standard. Les flux dynamiques restent plus performants que l'usage standard des sockets sur Myrinet (figure 10.5).

Nous avons effectué une évaluation du système de communication, dans le cadre de l'environnement d'exécution communicant MPI (MPICH 1.2.5.6). La figure 10.6 montre une comparaison entre la version MPI de l'application NetPipe, exécutée sur une grappe exploitée avec un système Linux classique, et la même application exécutée

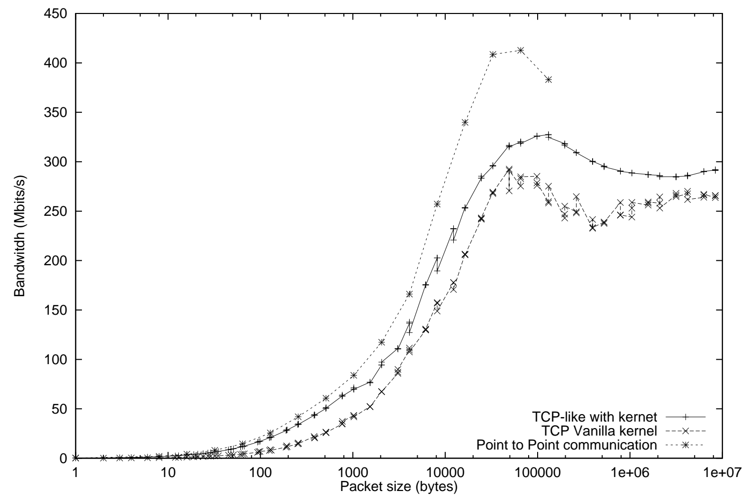


FIG. 10.4 – Performance sur Gigabit Ethernet

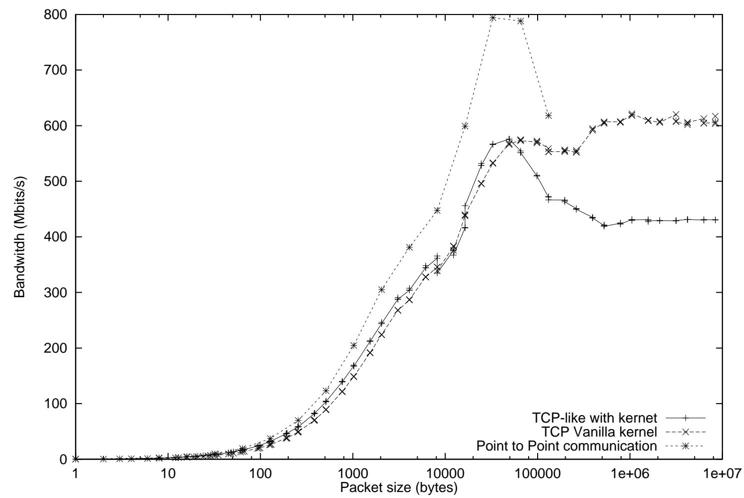


FIG. 10.5 – Performance sur Myrinet (netdevice interface)

avec Kerrighed. Notons que les logiciels MPICH et NetPipe restent inchangés dans la configuration Kerrighed par rapport à la configuration Linux standard (pas de recompilation). Dans le contexte de Kerrighed, l'environnement MPI est configuré comme pour une exécution sur une seule machine (le fichier de configuration déterminant la liste des machines se réduit à une ligne `localhost`). Nous exploitons le mécanisme de gestion globale des processeurs pour effectuer le placement des processus sur l'ensemble des nœuds de la grappe de calculateurs. Dans Kerrighed, le mécanisme des flux dynamiques gère l'ensemble des échanges de message et n'entraîne aucun surcoût pour l'application.

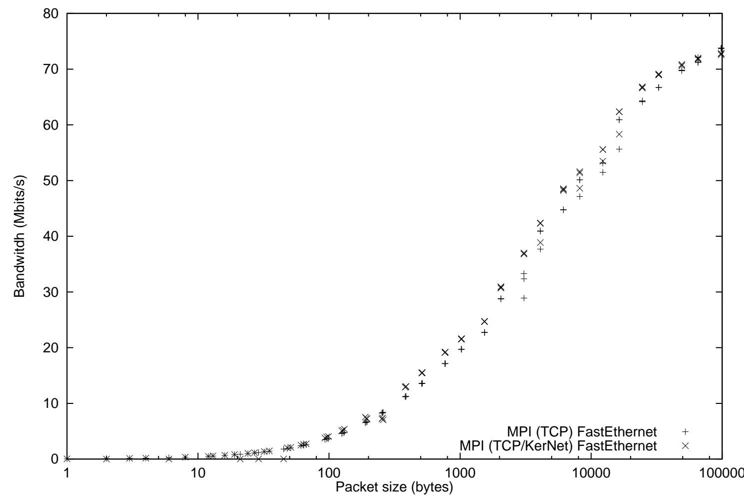


FIG. 10.6 – Performance MPI sur FastEthernet

Dans le cadre d'une collaboration avec la DGA, l'application GORPH3D a été mise à notre disposition. GORPH3D est un noyau de calcul en électromagnétisme, écrit pour l'environnement de communication MPI (en langage Fortran et en langage C). Pour cette évaluation, une grappe de 4 nœuds interconnectée par un réseau FastEthernet a été utilisée. Une comparaison entre une exécution au sein de Kerrighed et sans Kerrighed a été réalisée. Les temps d'exécution de cette application, pour différents jeux de données, se sont avérés jusqu'à 10% meilleurs avec Kerrighed. Ce résultat confirme le faible coût de notre infrastructure pour les applications parallèles par échange de messages.

Déplacement d'une extrémité de communication Un des objectifs des *flux dynamiques* est de permettre des échanges de message efficaces après le déplacement d'une extrémité de communication. Nous avons réalisé une telle évaluation avec l'application NetPipe. Cette application est déployée sur deux nœuds. Une des extrémités est déplacée vers un troisième nœud. Les mesures sont effectuées après le déplacement de cette extrémité. La figure 10.7 atteste de l'absence de coût, pour le flux, du déplacement d'une de ses extrémités.

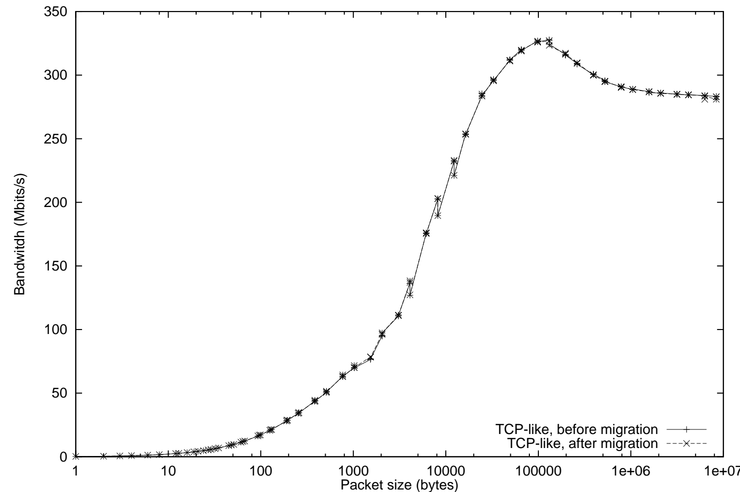


FIG. 10.7 – Performance sur Gigabit Ethernet lors d'un déplacement de processus

10.3 Évaluation comparative des performances de communication entre plusieurs SSIs

Une comparaison des mécanismes de flux que nous proposons, avec ceux offerts par d'autres systèmes à image unique (OpenSSI[40] et openMosix[1]), a été réalisée par Benoît Boissinot[54] dans le cadre d'un stage en 2004. Ces comparaisons ont été réalisées sur le même ensemble de machines, en exploitant un réseau d'interconnexion FastEthernet. L'application de test est une version de NetPIPE adaptée aux communications par *pipe*. Lors de l'exécution de l'application, les processus communicants se situent initialement sur le même nœud.

Des mesures de bande passante sont réalisées, pour différentes tailles de messages, dans quatre situations différentes :

- sans déplacement des processus,
- après le déplacement d'un des processus communicants,
- après le déplacement, sur deux nœuds distincts, des deux processus communicants,
- après le déplacement vers le même nœud des deux processus communicants.

La figure 10.8 présente les résultats sans déplacement. Les bandes passantes obtenues sont similaires pour l'ensemble des systèmes à image unique testés. Elles correspondent à la bande passante mémoire disponible sur le nœud concerné.

Les figures 10.9 et 10.10 présentent, respectivement, les mesures de bande passante après le déplacement d'un, puis des deux processus communicants. Une baisse de ces bandes passantes est notée pour les trois systèmes à image unique. En effet, les communications sont physiquement limitées par le réseau d'interconnexion. Le mécanisme de flux dynamique permet d'obtenir des résultats proches de la bande théorique du réseau FastEthernet (100Mbits) que ce soit à la suite d'un ou de deux déplacements

de processus. En revanche, le mécanisme de relai employé par OpenSSI et openMosix pénalise fortement les communications.

La figure 10.11 représente une situation où deux processus communicants se retrouvent sur un même nœud après migration. Une telle situation peut être le résultat d'une politique d'ordonnancement globale des processus visant à réduire l'usage du réseau. Dans une telle situation, le rapprochement de deux processus communicants vise à bénéficier de la bande passante de la mémoire physique du nœud hôte. Cependant, l'usage du mécanisme de relai retire tout le bénéfice d'une telle stratégie. À l'opposé, Kerrighed permet d'exploiter la plus grande bande passante disponible en employant un mécanisme de communication interne au nœud.

L'évaluation, selon le même protocole de test, des systèmes de communication par *socket unix* ou *socket inet* conduit aux mêmes résultats[54].

10.4 Synthèse

Nous avons présenté dans ce chapitre une évaluation des performances de l'infrastructure de communication réalisée au sein de Kerrighed. En particulier, nous nous sommes attachés à démontrer la pertinence et la flexibilité de notre approche dans la gestion des flux de données internes à une grappe de calculateurs ainsi que dans celle des services systèmes distribués.

Les transactions de communication permettent la conception de service système distribué réactif indépendamment de la charge du nœud concerné. Une telle propriété permet d'envisager la réalisation de mécanisme apte à réparer un nœud difficilement accessible par des moyens classiques reposant sur les processus (cas des dénis de services, d'une inssuffisance en ressource memoire obligeant le système à fortement sollicité sa zone d'échange temporaire sur disque, *etc.*).

Par ailleurs, les évaluations réalisées montrent que le mécanisme de flux dynamique permet l'exécution d'applications communiquant par échange de messages sur un système à image unique sans forte dégradation des performances. Ce mécanisme permet l'usage de différents types de réseau d'interconnexion. Par ailleurs, plusieurs interfaces de communication (*pipe*, *socket unix*, *socket inet*) ont été portées avec succès sur les flux dynamiques. Dans chacun de ces cas, les performances de l'interface de communication correspondent à celles du flux dynamique lui-même.

Une application parallèle écrite pour l'environnement MPI a été exécutée avec succès sur Kerrighed, sans dégradation des performances. À l'instar des systèmes à image unique openMosix et OpenSSI, Kerrighed ne nécessite aucune modification (du code source ou même du binaire) de l'application ou de l'environnement de communication afin de permettre l'exécution de l'application. Toutefois, grâce à l'usage combiné des mécanismes de création distante de processus et des flux dynamiques, il n'est pas nécessaire de décrire complètement la grappe afin de bénéficier de bonnes performances dans les communications. En effet, la configuration se résume simplement à indiquer un déploiement des processus sur *localhost* : la notion de nœuds d'une grappe est effacée au profit de la notion de système à image unique.

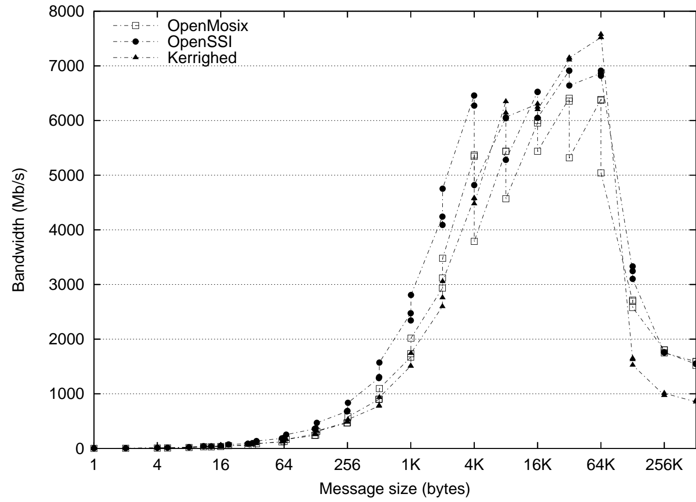


Fig. 10.8 – Bande passante des *pipe* sans déplacement

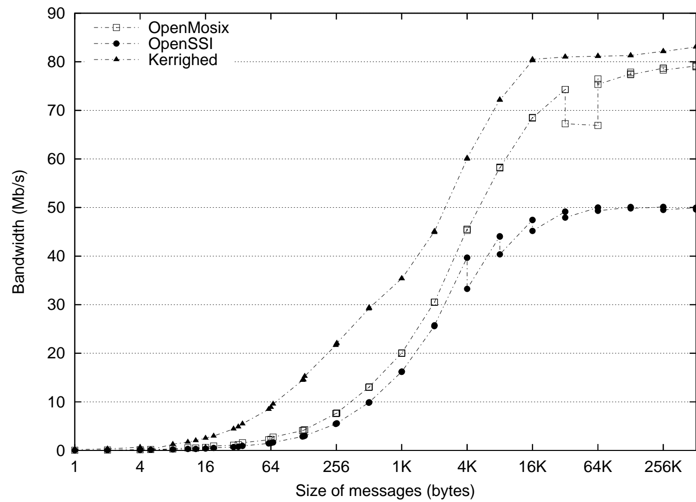


Fig. 10.9 – Bande passante des *pipe* après un déplacement

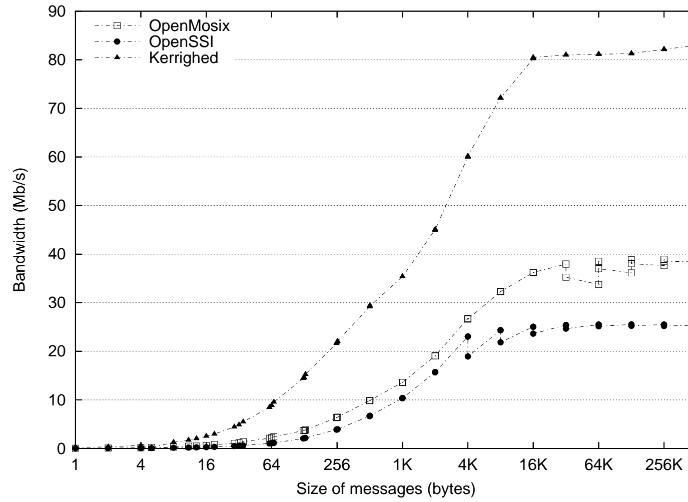


FIG. 10.10 – Bande passante des *pipe* après deux déplacements

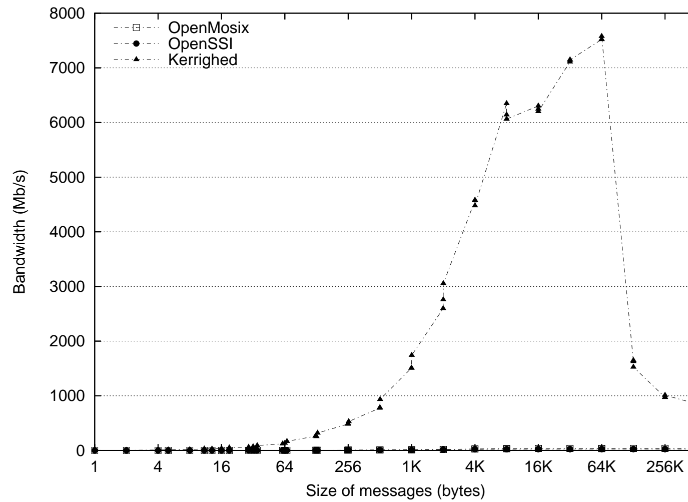


FIG. 10.11 – Bande passante des *pipe* après deux déplacements vers le même nœud

Enfin, comme attendu, les communications directes entre un processus émetteur et un processus récepteur offrent de meilleures performances que celles réalisées par un mécanisme de relai. Toutefois, le mécanisme de flux dynamique testé ne permet que des communications internes au système à image unique, alors que le mécanisme de relai permet des communications entre un processus de la grappe et un processus à l'extérieur de la grappe.

Conclusion

Bilan

Le système de communication occupe une place essentielle pour la performance et les fonctionnalités d'un système d'exploitation distribué pour grappe ainsi qu'une place essentielle pour les performances des applications exécutées sur cette grappe.

L'étude de la littérature montre qu'il existe de nombreux travaux tirant le meilleur profit de telle ou telle technologie réseau et notamment des réseaux à haut débit. Par ailleurs, il existe de multiples systèmes de communication spécifiques à une technologie, améliorant souvent les performances au prix de contraintes fortes (par exemple, un usage exclusif de l'interface de communication).

Plus généralement, de nombreux travaux ont été réalisés sur les environnements de communication par échange de message. Les performances de ces environnements sont fréquemment déterminées par l'exploitation d'une des primitives de communication de haute-performance, précédemment évoquées. De même, certains environnements de communication ont été spécialement adaptés pour offrir des propriétés telles que la transparence du déplacement de processus communicants ou la tolérance aux fautes des applications parallèles fondées sur l'échange de messages.

Cependant, il n'existe pas à notre connaissance de système permettant simultanément de répondre aux exigences de performance, et offrant un support au déplacement de processus communicants ainsi qu'un support de tolérance aux fautes. De plus, aucun système de communication n'adresse spécifiquement le problème des communications de système d'exploitation à système d'exploitation pour les besoins internes du système d'exploitation. En effet, ce besoin n'était pas identifié avant l'apparition des systèmes à image unique. De plus, les systèmes à image unique existants éludent la question en employant un modèle de communication inter-processus traditionnel.

Nous avons proposé un système de communication complet pour systèmes à image unique sur des grappes de calculateurs destinées à l'exécution d'applications à haute performance.

Un système à image unique tel que Kerrighed repose sur un ensemble de services système distribués. Une première partie de notre contribution est de fournir un modèle de conception des services distribués. Un service système distribué est défini par un modèle d'invocation de fonction à distance. Les communications sont liées aux appels de fonction ou à leur réponse. Les **transactions de communication** permettent de définir précisément la construction ou l'interprétation d'un message, et permettent de

spécifier les traitements possibles à la suite d'une réception de message. Les transactions de communication permettent de composer les messages ainsi que les actions découlant de ces communications.

Le deuxième aspect d'un système de communication pour système à image unique, est le support efficace des applications communiquant par échange de messages. Les **flux dynamiques** apportent une solution au problème de la localisation d'une extrémité de communication ainsi qu'à celui de la capture de l'état d'un canal de communication. Grâce aux flux dynamiques, les échanges de données entre processus se font directement entre le nœud émetteur et le nœud récepteur, sans l'intervention coûteuse de nœuds intermédiaires. Les flux dynamiques constituent un modèle d'interface mobile pour système à image unique. L'ensemble des interfaces de communication standard ou des bibliothèques de communication de bas niveau peut être adapté pour fournir les propriétés nécessaires à l'exécution des applications haute performance dans le cadre d'un système à image unique.

L'architecture proposée est extensible. Elle permet l'ajout de nouveaux protocoles ou de nouvelles interfaces de communication. Une abstraction du matériel permet une grande portabilité sur les différentes technologies de réseaux. Au final, cette architecture est applicable à d'autres systèmes que celui retenu pour notre prototype.

Kerrighed est le système à image unique que nous avons employé comme plateforme expérimentale. L'architecture complète du système de communication KerNet, que nous avons proposé, a progressivement été intégrée à Kerrighed. Les transactions de communication sont à la base de l'ensemble des communications liées aux services distribués de Kerrighed. Le modèle de service distribué a été employé pour réaliser le service de flux dynamique permettant le déplacement et la sauvegarde de points de reprise de processus communiquant par échange de messages. Un support aux interfaces de communication de type pipe, socket unix et socket inet a été mis en œuvre. Le prototype ainsi réalisé a été validé par le déploiement et l'exécution, en présence d'une politique d'ordonnancement d'équilibrage de charge dynamique, d'applications reposant sur l'environnement de programmation MPI. La propriété de transparence complète (tant au niveau source des applications qu'au niveau binaire) a ainsi pu être vérifiée sur des applications d'origine industrielle fournies par EDF R&D, l'ONERA et par la DGA.

Les deux mécanismes proposés visent à améliorer les performances des applications s'exécutant sur un SSI. Ainsi, les transactions de communication permettent une amélioration de la réactivité des services système du SSI. Par ailleurs, les flux dynamiques permettent le déplacement de processus communicants sans dégradation de leurs performances.

Ce travail, combiné à celui de Renaud Lottiaux pour la gestion globale de la mémoire[55] et du système de fichiers, et à celui de Geoffroy Vallée pour la gestion globale des processus[96] au sein du SSI Kerrighed, permet de disposer d'un SSI complet vis-à-vis de l'ensemble des ressources présentes dans une grappe de calculateurs.

Le système de communication KerNet, intégré au système à image unique Kerrighed, est disponible sur le site web de Kerrighed (*http ://www.kerrighed.org*). Ce système est distribué, comme l'intégralité de Kerrighed, sous forme de logiciel libre avec une licence

GPL.

L'objectif d'un système à image unique étant d'allier la performance des applications à la simplicité d'utilisation de la grappe de calculateurs, il est démontré que le surcoût de notre architecture de communication est faible relativement aux propriétés apportées. Déployer et déplacer des processus communiquant par échange de messages reste donc efficace dans les systèmes à image unique disposant d'un système de communication adapté.

Perspectives

Au sein d'une grappe de calculateurs, les besoins en communication sont multiples. Le travail présenté établit, dans le cadre des applications haute performance, la nécessité de disposer d'un système de communication adapté aux systèmes à image unique. Plusieurs perspectives se dégagent de nos travaux.

Perspectives à court terme Le prototype actuel du système de communication proposé met en œuvre toute l'infrastructure nécessaire (abstraction du matériel, gestion des protocoles de communication, transaction de communication et flux dynamique). Cependant, notre prototype n'est pas encore pleinement intégré dans Kerrighed. En effet, la plupart des services existants dans Kerrighed utilisent une version simplifiée des transactions (généralement le modèle *send&forget* avec réception bloquante). L'usage complet des transactions de communication dans des services tels que la gestion globale de la mémoire ou celui de la gestion globale des processus devrait tirer profit directement de ces nouveaux modèles de communication.

Par ailleurs, seuls les flux correspondants aux interfaces socket et pipe ont été complétement mis en œuvre et évalués. Le prototype actuel doit donc être complété afin de prendre en charge des interfaces de communication avec accès direct au matériel tel que nous le décrivons dans le paragraphe 8.2.5.2. Enfin, une dernière classe de flux n'a pas été abordée dans ce travail. Il s'agit des périphériques de type caractère présents sur l'ensemble des systèmes Unix. Ces périphériques comprennent notamment l'accès aux consoles pour les applications. L'ajout d'une propriété permettant de déplacer une extrémité d'un tel flux doit être envisagé afin d'améliorer la transparence du mécanisme de SSI aux applications ainsi qu'aux usagers.

Le modèle de matériel de communication ainsi que celui des protocoles de communication peuvent être étendus, notamment en améliorant les mécanismes de contrôle de flux. En effet, le mécanisme de contrôle de flux actuel permet à un service système distribué de réaliser un déni de service sur un autre nœud de la grappe, simplement en le surchargeant de messages. Dans un futur contexte de haute disponibilité, des mécanismes de contrôle de flux plus fins doivent être mis en place pour se prémunir de ce type de problème.

Lors de l'exécution d'une application parallèle communiquant par message, il arrive que les charges processeur des processus composant cette application, ne soient pas

toutes égales. L'ordonnanceur global actuel de Kerrighed sait déplacer les processus de manière à répartir au mieux les charges. Cependant, certaines décisions peuvent se retrouver en contradiction avec l'intensité des communication entre deux processus. Ainsi deux processus calculant peu mais s'échangeant une grande quantité de données devraient être placés sur le même nœud. Par conséquent, une extension de l'ordonnanceur actuel ainsi que la conception de politiques de placement et de déplacement de processus adaptées sont souhaitables.

Finalement, l'architecture de Kerrighed étant désormais complète, il nous est possible d'évaluer l'ensemble de Kerrighed dans des domaines encore peu explorés par lui :

- l'évaluation des performances d'applications industrielles complètes (et plus seulement de noyaux de calcul industriel),
- l'évaluation de Kerrighed dans le domaine des serveurs réseau (serveurs de base de données ou serveurs web),
- la comparaison de différentes stratégies de placement de processus et de sauvegarde de point de reprise d'applications parallèles communiquant par échange de messages,
- l'évaluation et la comparaison des trois modèles de programmation d'applications parallèles les plus fréquents : mémoire partagée, échange de messages, application hybride (mémoire partagée et échange de messages).

Perspectives à moyen et long terme L'architecture de communication présentée a eu pour principal objectif d'améliorer les performances au sein d'un SSI. Cette approche du système de communication reste traditionnelle par rapport à ce qui se fait dans la plupart des intergiciels de communication. Le système de communication est utilisé pour échanger des données à la demande des machines hôtes.

L'arrivée d'interfaces de communication 'intelligentes', dotées de leur propre mémoire et de leur propre processeur embarqué leur conférant une certaine autonomie, permet d'envisager le système de communication comme un bus système à l'échelle de la grappe de calculateurs. Un tel bus, véritable prolongement des bus système internes à chacun des nœuds, permettrait d'offrir en plus des propriétés de performance, les propriétés de haute disponibilité nécessaires à la réalisation d'un véritable système à image unique hautement disponible.

Une première approche pour améliorer les performances de communication serait de substituer au processeur central de la machine hôte, le processeur embarqué de la carte de communication. En effet, les mécanismes d'interruption matérielle sont associés à un coût (en terme de temps) généralement élevé. Lors d'opérations très simples telles que l'incrémentation d'un compteur d'utilisation de page, le coût du traitement par le système de communication reste encore très élevé par rapport à celui de l'opération effectuée. De telles opérations pourraient bénéficier d'un traitement réalisé par le processeur de communication lui-même. Cependant, il ne faut pas perdre de vue que ce processeur embarqué est généralement beaucoup moins puissant que le processeur hôte. Un équilibre doit donc être déterminé entre les tâches réalisées par le processeur central et celles prises en charge par le processeur de communication.

Une deuxième forme d'utilisation du processeur embarqué sur les cartes d'interface réseau concerne la haute disponibilité de la grappe. Dans tout système informatique, un ensemble de mécanismes de surveillance des noeuds doit être mis en œuvre. Or, de tels mécanismes alourdissent la charge des machines et peuvent être corrompus. Dans la continuïté des travaux réalisés à Rutgers University, sous la direction de L. Iftode, il est possible de répartir le travail de surveillance sur les interfaces de communication. De plus, il a été démontré que des opérations simples pour corriger un problème tel que le *fork bomb* qui vise à saturer les ressources d'un nœud en créant autant de processus que possible, sont efficaces alors que le nœud est considéré comme "mort". Par conséquent, l'usage d'un ensemble de processeurs embarqués sur les interfaces de communications et indépendantes du système d'exploitation semble être une perspective possible pour une approche originale de la haute-disponibilité au sein des systèmes à image unique. Un tel travail permettrait d'ouvrir une voie pour une étude précise des problèmes liés au défaillance au sein d'un SSI ainsi que des changements de configurations (ajout ou retrait de nœuds volontaire) au sein de la grappe.

Enfin, il serait intéressant d'étudier, de manière concrète, l'intégration du système de communication KerNet au sein d'autres systèmes à image unique tels que OpenMosix ou OpenSSI. En effet, une telle réalisation a du sens car ces systèmes doivent aussi déployer des services système distribués basés sur les appels de procédure distante et visent à exécuter efficacement des applications parallèles conçues sur le modèle de l'échange de messages.

Bibliographie

[1] http ://openmosix.sourceforge.net/.

[2] MICO, an OpenSource CORBA implementation. *http ://www.mico.org/.*

[3] The Object Management Group. World Wide Web document, *http ://www.omg.org*.

[4] A. Agbaria and R. Friedman. Starfish : Fault-tolerant dynamic mpi programs on clusters of workstations. In Eighth IEEE Int'l Symp. on High Performance Distributed Computing, pages 167–176, Août 1999.

[5] Lior Amar, Amnon Barak, and Amnon Shiloh. The MOSIX Parallel I/O System for Scalable I/O Performance. In Proc. 14-th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002), pages 495–500, Cambridge, MA, Novembre 2002.

[6] Gabriel Antoniu, Luc Boug, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99), volume 1586 of Lect. Notes in Comp. Science, pages 496–510, San Juan, Porto Rico, 1999. Held in conjunction with IPPS/SPDP 1999. IEEE TCPP and ACM SIGARCH, Springer-Verlag.

[7] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM : A single system image of a JVM on a cluster. In International Conference on Parallel Processing, pages 4–11, 1999.

[8] InfiniBand Trade Association. Infiniband architecture specification. Technical report, InfiniBand Trade Association, Novembre 2002.

[9] OmniORB Home Page. AT&T Laboratories Cambridge, *http ://www.omniorb.org*.

[10] Olivier Aumage. Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes. PhD thesis, École Normale Supérieure de Lyon, 2002.

[11] Olivier Aumage, Luc Bougé, Alexandre Denis, Jean-François Méhaut, Guillaume Mercier, Raymond Namyst, and Loïc Prylli. Madeleine II : a Portable and Efficient Communication Library for High-Performance Cluster Computing. In Proceeding of the IEEE International Conference on Cluster Computing

(CLUSTER'00), pages 78–87, Chemnitz, Allemagne, 28 Novembre - 01 Décembre 2000.

[12] Ramamurthy Badrinath, Christine Morin, and Geoffroy Vallée. Checkpointing and recovery of shared memory parallel applications in a cluster. In Proc. Intl. Workshop on Distributed Shared Memory on Clusters (DSM 2003), pages 471–477, Tokyo, May 2003. Held in conjunction with CCGrid 2003. IEEE TFCC. Revised version of [13].

[13] Ramamurthy Badrinath, Christine Morin, and Geoffroy Vallée. Checkpointing and recovery of shared memory parallel applications in a cluster. Research Report RR-4806, INRIA, IRISA, Rennes, France, April 2003. Preliminary version of [12].

[14] Amnon Barak, Shai Guday, and Richard G. Wheeler. The MOSIX Distributed Operating System, Load Balancing for UNIX, volume 672 of Lecture Notes in Computer Science. Springer-Verlag, 1993.

[15] Jim Basney and Miron Livny. Managing network resources in Condor. In Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9), pages 298–299, Pittsburgh, PA, Août 2000.

[16] Rajanikanth Batchu, Jothi P. Neelamegam, Zhenqian Cui, Murali Beddhu, Anthony Skjellum, Yoginder Dandass, and Manoj Apte. MPI/FT : Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing. In Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID'01), pages 26–33, Brisbane, Australia, Mai 2001.

[17] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In Proceedings of the ACM Symposium on Operating System Principles, page 3, Bretton Woods, NH, 1983. Association for Computing Machinery.

[18] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet : A gigabit-per-second local area network. IEEE Micro, 15(1) :29–36, 1995.

[19] G. Bosilca, A. Bouteiller, F. Cappello, S Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V : Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In IEEE Proc. of SuperComputing, 2002.

[20] Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. In Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'03), 2003.

[21] Greg Burns, Raja Daoud, and James Vaigl. LAM : An Open Cluster Environment for MPI. In Proceedings of Supercomputing Symposium, pages 379–386, 1994.

[22] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report Technical Report ANL-92/17, Argonne National Laboratory, Octobre 1992.

[23] Don Cameron and Greg Regnier. The Virtual Interface Architecture. INTEL Press, Avril 2002.

[24] Jeremy Casas, Dan L. Clark, Ravi Konuru, Steve W. Otto, Robert M. Prouty, and Jonathan Walpole. MPVM : A migration transparent version of PVM. Usenix Computing Systems, 8(2) :171–216, 1995. http ://www.cse.ogi.edu/DISC/projects/mist/.

[25] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive : fault containment for shared-memory multiprocessors. In Proceedings of the fifteenth ACM symposium on Operating systems principles, pages 12–25. ACM Press, 1995.

[26] Giuseppe Ciaccio. Optimal communication performance on Fast Ethernet with GAMMA. In IPPS/SPDP Workshops, volume LNCS 1388, pages 534–548. Springer-Verlag, 1998.

[27] Guiseppe Ciaccio and Giovanni Chiola. Low-cost gigabit ethernet at work. In International Conference on Cluster Computing (CLUSTER'00), pages 359–360, Chemnitz, Germany, Novembre 2000.

[28] Frame Relay Forum Technical Committee. Pvc user-to-network interface (uni) implementation agreement. Technical report, Frame Relay Forum, Juillet 2000.

[29] Antonio F. Díaz, Jesús Ferreira, Julio Ortega, Antonio Cañas, and Alberto Prieto. CLIC : Fast Communication on Linux Clusters. In International Conference on Cluster Computing (CLUSTER'00), pages 365–366, Chemnitz, Germany, Novembre 2000.

[30] Message Passing Interface Forum. MPI : A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, 1994.

[31] Message Passing Interface Forum. MPI-2 : Extension to the Message-Passing Interface. Technical report, University of Tennessee, 1997.

[32] Roy Friedman, Maxim Goldin, Ayal Itzkovitz, and Assaf Schuster. MILLIPEDE : Easy parallel programming in available distributed environments. Software Practice and Experience, 27(8) :929–965, 1997.

[33] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix : A Global Layer Unix for a network of workstations. Software – Practice and Experience, 28(9) :929–961, 1998.

[34] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B.D. Johnsen, H. Kohmann, R. Nordstrøm, and P. Werner. Low-Level SCI software functional specification. Esprit Project 23174, Mars 1999. Version 2.1.1.

[35] R. Goeckelmann, M. Schoettner, S. Frenz, and P. Schulthess. A kernel running in a dsm - design aspects of a distributed operating system. In Proceedings of the IEEE International Conference on Cluster Computing, pages 478–482, Hong-Kong, 2003.

[36] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification. Sun Microsystems, Inc., 2000. Second edition.

[37] Yvon J gou. Implementation of page management in Mome, a user-level DSM. In Proc. Intl. Workshop on Distributed Shared Memory on Clusters (DSM 2003), pages 479–486, Tokyo, Japon, 2003. Held in conjunction with CCGrid 2003. IEEE TFCC.

[38] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. Technical report, Argonne National Laboratory, 2004.

[39] Ralf Große, Röger Butenuth, and Hans-Ulrich Heiß. IP over SCI. In Proceeding of the IEEE International Conference on Cluster Computing (CLUSTER'00), pages 73–77, Chemnitz, Allemagne, 28 Novembre - 01 Décembre 2000.

[40] OpenSSI group. Introduction to the ssi cluster. Technical report, OpenSSI, 2003.

[41] M. Hayden. The ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, Janvier 1998.

[42] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, Texas, Octobre 2003.

[43] Masanori Itoh, Tooru Ishizaki, and Mitsuhiro. Accelerated Socket Communication in System Area Networks. In Proceeding of the IEEE International Conference on Cluster Computing (CLUSTER'00), pages 357–358, Chemnitz, Allemagne, 28 Novembre - 01 Décembre 2000.

[44] R. Jain. Fddi : Current issues and future trends. IEEE Communications Magazine, pages 98–105, Septembre 1993.

[45] Povl T. Koch, J. S. Hansen, E. Cecchet, and X. Rousset de Pina. Scios : An sci-based software distributed shared memory. In Proceedings of the 1st Workshop on Software Distributed Shared Memory, pages 20–25, 1999.

[46] Ravi B. Konuru, Steve W. Otto, and Jonathan Walpole. A migratable user-level process package for PVM. Journal of Parallel and Distributed Computing, 40(1) :81–102, 1997.

[47] M. E. Kounavis, A. T. Campbell, S. Chou, F. Modoux, J. Vicente, , and H. Zhang. The genesis kernel : A programming system for spawning network architectures. IEEE Journal on Selected Areas in Communications (JSAC), 19 :49–73, Mars 2001.

[48] Orion Sky Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. Concurrency and Computation : Practice and Experience, 15 :371–393, 2003.

[49] Jean-Yves Le Boudec. The asynchronous transfer mode : a tutorial. Comput. Netw. ISDN Syst., 24(4) :279–309, 1992.

[50] Juan Leon, Allan L. Fisher, and Peter Steenkiste. Fail-safe PVM : A Portable Package for Distributed Programming with Transparent Recovery. Technical Report CMU-CS-93-124, Février 93.

[51] Kai Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD thesis, Yale University, 1986.

[52] Michael Litzkow and Miron Livny. Supporting checkpointing and process migration outside the UNIX kernel. In Proceedings of the Winter 1992 USENIX Conference, pages 283–290, San Francisco, CA, Janvier 1992.

[53] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, Avril 1997.

[54] R. Lottiaux, B. Boissinot, and Christine Morin. Openmosix, openssi and kerrighed : a comparative study. Rapport de recherche PI-1656, IRISA, IRISA, Rennes, France, November 2004.

[55] Renaud Lottiaux. Gestion globale de la mémoire physique d'une grappe pour un système à image unique : mise en œuvre dans le système Gobelins. PhD thesis, IRISA, Université de Rennes 1, 2001.

[56] David Margery, Geoffroy Vall e, Renaud Lottiaux, Christine Morin, and Jean-Yves Berthou. Kerrighed : a SSI cluster OS running OpenMP. In Proc. 5th European Workshop on OpenMP (EWOMP '03), 2003. Also published as [57].

[57] David Margery, Geoffroy Vall e, Renaud Lottiaux, Christine Morin, and Jean-Yves Berthou. Kerrighed : a SSI cluster OS running OpenMP. Research Report RR-4947, INRIA, IRISA, Rennes, France, 2003. Now published as [56].

[58] Christine Morin and Renaud Lottiaux. Global resource management for high availability and performance in a dsm-based cluster. In Proc. of 1st workshop on Software Distributed Shared memory, 1999.

[59] Myricom. The GM-1 Message Passing System. http ://www.myri.com/scs.

[60] Myricom. The GM-2 Message Passing System. http ://www.myri.com/scs.

[61] Myricom. Myrinet Express (MX) : A High Performance, Low-Level, Message-Passing Interface for Myrinet. http ://www.myri.com/scs.

[62] Myricom. Performance of mpich-gm 1.2.4..8a uniprocessor (up) case (one process per node). Technical report, Myricom, 2004.

[63] CORPORATE Institute of Electrical and Inc. Staff Electronics Engineers. IEEE Standard for Scalable Coherent Interface, Science : IEEE Std. 1596-1992. IEEE Standards Office, 1993.

[64] OMG. The Common Object Request Broker : Architecture and Specificat ion (Revision 2.5). OMG Document formal/01-09-34, 2001.

[65] John K. Ousterhout, A. R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. Computer, 21(2) :23–36, 1988.

[66] Ross A. Overbeek and James Boyle. Portable Programs for Parallel Processors. Saunders College Publishing, 1987.

[67] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network : High Performance Clustering Technology. IEEE Micro, 22(1) :46–57, Janvier-Février 2002.

[68] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt** : Transparent checkpointing under Unix. In Usenix Winter Technical Conference, pages 213–223, Janvier 1995.

[69] Linux Infiniband Project. Linux system software for the infiniband architecture. Technical report, -, 2002. http ://infiniband.sourceforge.net/LinuxSAS.1.0.1.pdf.

[70] L. Prylli and Bernard Tourancheau. BIP : a New Protocol Design for High Performance Networking on Myrinet. In Sringer-Verlag, editor, 1st Workshop on Personnal Computer based Networks Of Workstations (PC-NOW'98), volume 1388 of Lecture Notes in Computer Science, pages 472–485. IEEE, 1998. Held in conjunction with IPPS/SPDP 1998.

[71] L. Prylli, Bernard Tourancheau, and Roland Westrelin. An Improved NIC program for high-performance MPI. In Workshop on Cluster-Based Computing, Internation Conference in SuperComputing, pages 26–30, Rhodes, Grèce, Juin 1999. ACM.

[72] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16), 1997.

[73] Douglas Schmidt, Aniruddha Gokale, Timothy Harrison, and Guru Parulkar. A high-performance endsystem architecture for real-time COR BA. IEEE Communication Magazine, 14(2), 1997.

[74] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVI), pages 297–307, 1994.

[75] Anton Selikhov, George Bosilca, Cécile Germain, Gilles Fédak, and Franck Cappello. MPICH-CM : A Communication Library Design for a P2P MPI Implementation. In EuroPVM/MPI 2002, 2002.

[76] Tom Shanley. PCI-X System Architecture. Addison Wesley, 2003.

[77] Tom Shanley and Don Anderson. PCI System Architecture. Addison Wesley, 2003.

[78] John F. Shoch. An introduction to the ethernet specification. SIGCOMM Comput. Commun. Rev., 11(3) :17–19, 1981.

[79] Q. Snell, A. Mikler, and J. Gustafson. NetPIPE : A Network Protocol Independent Performace Evaluator. In IASTED International Conference on Intelligent Information Management and Systems, Juin 1996.

[80] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In Proceedings, 10th European PVM/MPI Users' Group Meeting, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venise, Italie, Septembre-Octobre 2003. Springer-Verlag.

[81] Georg Stellner. CoCheck : Checkpointing and Process Migration for MPI. In Proceedings of the 10th International Parallel Processing Symposium (IPPS'96), Honolulu, Hawaii, Avril 1996.

[82] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF : A parallel workstation for scientific computation. In Proceedings of the 24th International Conference on Parallel Pr ocessing, pages I :11–14, Oconomowoc, WI, 1995.

[83] Richard Stevens. Unix Network Programming, volume Volume 1 : Networking APIs : Sockets and XTI. Prentice Hall, 1998.

[84] Richard Stevens. Unix Network Programming, volume Volume 2 : Interprocess Communications. Prentice Hall, 1999.

[85] W. Richard Stevens. TCP/IP Illustrated, volume Volume 1 : The Protocols. Addison-Wesley, 1994. ISBN 0-201-63346-9.

[86] W. Richard Stevens. TCP/IP Illustrated, volume Volume 2 : the Implementation. Addison-Wesley, 1995. ISBN 0-201-63354-X.

[87] W. Richard Stevens. TCP/IP Illustrated, volume Volume 3 : TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols. Addison-Wesley, 1996. ISBN 0-201-63495-3.

[88] F. Sultan, A. Bohra, P. Gallard, I Neamtiu, S. Smaldone, Y. Pan, and L. Iftode. Recovering internet services from operating system failures. IEEE Internet Computing, March/April 2005. To appear.

[89] F. Sultan, A. Bohra, I. Neamtiu, and L. Iftode. Nonintrusive remote healing using backdoors. In Proceedings of First Workshop on Algorithms and Architectures for Self-Managing Systems, in conjunction with ISCA '03, San Diego, CA, Juin 2003.

[90] Florin Sultan. System Support for Service Availability, Remote Healing and Fault Tolerance using Lazy State Propagation. PhD thesis, Rutgers, State University of New Jersey, Octobre 2004.

[91] Inc. Sun Microsystems. Nfs : Network file system protocol specification. Technical report, IETF, Mars 1989.

[92] V. S. Sunderam. PVM : a framework for parallel distributed computing. Concurrency, Practice and Experience, 2(4) :315–340, 1990.

[93] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (Network Of Workstation). IEEE Micro, 15(1) :54–64, February 1995.

[94] Bernard Tourancheau and Roland Westrelin. Study of Medium Message Performance of BIP/Myrinet. In Proceeding of the IEEE International Conference on Cluster Computing (CLUSTER'00), pages 65–72, Chemnitz, Allemagne, 28 Novembre - 01 Décembre 2000.

[95] Geoffroy Vallée, Christine Morin, Jean-Yves Berthou, and Louis Rilling. A new approach to configurable dynamic scheduling in clusters based on single system image technologies. In Proc. of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), page 91, Nice, 2003. IEEE.

[96] Geoffroy Vallée. Conception dun ordonnanceur de processus adaptable pour la gestion globale des ressources dans les grappes de calculateurs : mise en œuvre

dans le système d'exploitation. PhD thesis, IRISA, Université de Rennes 1, Mars 2004.

[97] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net : a user-level network interface for parallel and distributed computing. In Proceedings of the fifteenth ACM symposium on Operating systems principles, pages 40–53. ACM Press, 1995.

[98] T. von Eiken, D. Culler, S. Goldstein, and K. Schauser. Active Messages : A Mechanism for Integrated Communication and Computation. In Proceeding of the 19th International Symposium on Computer Architecture, Gold Coast, Australie, Mai 1992. ACM Press.

[99] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Low-latency communication over fast ethernet. In Euro-Par, Vol. I, pages 187–194, 1996.

[100] M. Wende, M. Schoettner, O. Schirpf, and P. Schulthess. Network design for the DSM Operating System Plurix. In Proceedings of the 3rd Workshop "Kommunikationstechnik"/ IEEE Communications Society Chapter Germany, Juillet 1999.

Résumé

Dans la lignée des réseaux de stations de travail, les grappes de calculateurs représentent une alternative attrayante, en terme de performance et de coût, comparativement aux machines parallèles traditionnelles, pour l'exécution d'applications parallèles de calcul à haute performance. Une grappe de calculateurs est constituée d'un ensemble de nœuds interconnectés par un réseau dédié à haute performance. Les **systèmes à image unique** (*Single System Image* – **SSI**) forment une classe de logiciel offrant aux utilisateurs et programmeurs d'une grappe de calculateurs, l'illusion d'une machine unique. Un SSI peut être conçu à différents niveaux (intergiciel, système d'exploitation) selon le degré d'exigence quant à la réutilisation sans modification de modèles de programmation et d'applications existants. Dans notre contexte, les applications visées sont de type MPI ou OpenMP. Comme pour tout système distribué, le système d'interconnexion des nœuds de la grappe se trouve au cœur des performances globales de la grappe et des SSIs.

Les travaux présentés dans cette thèse portent sur la conception d'un **système de communication dédié aux systèmes d'exploitation distribués pour grappes**. Ces travaux s'inscrivent dans le cadre de la conception et la réalisation d'un SSI pour l'exécution d'applications haute performance sur grappe de calculateurs.

Notre première contribution se situe dans la conception d'un modèle de communication adapté aux communications internes aux services systèmes distribués qui constituent le SSI. En effet, de la performance des communications dépendent les performances globales de la grappe. Les **transactions de communication** permettent (`i`) de décrire un message lors de sa création, (`ii`) d'acheminer efficacement le message en fontion des ressources disponibles, et (`iii`) de délivrer et traiter le message au plus tôt sur le nœud destinataire.

Notre seconde contribution correspond à la conception d'un support au déplacement de processus communiquant par flux de données (*socket*, *pipe*, etc.). En effet, au sein d'un SSI, les processus peuvent être déplacés en cours d'exécution par un ordonnanceur global. Les **flux dynamiques** permettent le déplacement d'une extrémité de communication sans dégradation des performances.

Nos propositions ont été mises en œuvre dans le prototype de SSI Kerrighed, conçu au sein du projet INRIA PARIS de l'IRISA. Ce prototype nous a permis d'évaluer le système de communication proposé. Nous avons montré une réactivité accrue des services systèmes distribués ainsi qu'une absence de dégradation des performances des applications communiquant par messages (en particulier MPI) après déplacement d'un processus. L'ensemble de ce travail est distribué sous licence GPL en tant que partie de Kerrighed et est disponible à l'adresse *http ://www.kerrighed.org.*